NOUVEAUX PROGRAMMES

Jean-Noël Beury

# INFORMATIQUE AVEC PYTHON

## MPSI-PCSI-PTSI
## MP-PC-PSI-PT-TSI-TPC

j'intègre

## MÉTHODES & EXERCICES

- Toutes les méthodes
- 79 exercices repérés par difficulté
- Des extraits de sujets de concours
- Des indications pour démarrer la résolution
- Tous les corrigés rédigés

2e édition

LES + EN LIGNE

DUNOD

Jean-Noël Beury

# INFORMATIQUE AVEC PYTHON

## MPSI-PCSI-PTSI
## MP-PC-PSI-PT-TSI-TPC

j'intègre

## MÉTHODES & EXERCICES

2e édition

DUNOD

# Table des matières

# Avant-propos

## Présentation générale

Cet ouvrage de la série « Méthodes & Exercices » traite de l'intégralité du programme d'informatique commune aux classes préparatoires aux grandes écoles. Chacun des seize chapitres est divisé en quatre parties.

### Les méthodes à retenir

Chaque chapitre commence par des rappels de cours synthétiques, des méthodes de raisonnement avec des exemples de programmes.

### Énoncés des exercices

Des énoncés d'exercices d'application du cours et de nombreux exercices d'écrits et d'oraux de concours sont proposés. Ils sont affectés d'un niveau de difficulté, de 1 à 4.

### Du mal à démarrer ?

Des indications de méthode sont données pour le cas où cela vous serait nécessaire.

### Corrigés des exercices

Les solutions détaillées sont entièrement rédigées. De nombreux commentaires sont ajoutés dans les programmes Python, en particulier les indices de début et de fin pour les boucles `for`.

**« Plus en ligne »**

Vous pouvez télécharger à partir de la page de présentation de l'ouvrage sur le site Dunod tous les programmes Python et les requêtes SQL des exercices. Des fichiers complémentaires sont également fournis afin de tester les programmes, par exemple des images pour l'exercice sur la stéganographie dans le traitement des images. Pour le chapitre 15, des fichiers .txt sont fournis afin de créer facilement une base de données pour tester les requêtes SQL.


https://dunod.com/EAN/9782100841240

## Conseils de travail

Les deux premiers chapitres permettent de vous approprier la syntaxe Python. Je vous conseille de les relire régulièrement pour bien comprendre le langage Python : typage des variables, indentation, boucle, test, fonction, représentation graphique…

Avec une bonne maîtrise du langage Python, vous pouvez vous concentrer sur les consignes des problèmes de concours. Les énoncés demandent très souvent d'écrire des fonctions. Faites bien attention aux arguments d'entrée et arguments

de sortie. Pensez à utiliser les fonctions définies dans les questions précédentes. L'optimisation des algorithmes est souvent demandée afin de réduire la complexité.

À l'écrit, vous ne pouvez pas tester vos programmes Python sur un ordinateur. Je vous conseille de prendre une feuille de brouillon et de vérifier les différentes étapes réalisées par les boucles `for`, en particulier la première et la dernière. Les indices des listes ne doivent pas dépasser une valeur limite.

Soyez bien vigilant à l'indentation. Et n'oubliez pas que le programme quitte immédiatement une fonction quand il rencontre l'instruction `return`.

Entraînez-vous régulièrement à écrire des programmes Python et des requêtes SQL !

J'espère que cet ouvrage vous aidera à réussir le mieux possible l'épreuve d'informatique des concours et je vous souhaite bon courage pour votre travail.

# Types, listes, boucles, fonctions

CHAPITRE 1

Plan



### Thèmes abordés dans les exercices

- Types utilisés avec Python : `int`, `float`, `str`, `list`, `bool`, `tuple`, `deque`, `dict`
- Tests avec les opérateurs : `and`, `or`, `not`
- Bibliothèques et modules
- Boucles `for`, `while`
- Définition d'une fonction
- Variable locale, variable globale

### Points essentiels du cours pour la résolution des exercices

- Compteur, accumulateur
- Ajouter, supprimer des éléments dans une liste
- Création d'une liste par compréhension
- Slicing ou extraction de tranche
- Manipuler des listes de listes
- Bien identifier les indices de début et de fin dans les boucles
- Utiliser les fonctions définies précédemment dans d'autres fonctions

## Les méthodes à retenir

### Installation de Python 3

Il existe de très nombreux programmes permettant d'installer Python 3 sur l'ordinateur. On peut utiliser par exemple le logiciel Pyzo, qui permet de créer des programmes avec l'extension `.py`. Les programmes écrits avec Python 2 ne sont pas compatibles avec Python 3.

### Script Python

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.

# Types de base utilisés dans Python

### Types `int` (entier), `float` (flottant)

Pour affecter une valeur dans une variable, on utilise « = » :

```
a=3    # a prend la valeur 3
```

Cette instruction affecte 3 dans la variable `a`.

Le typage des variables est dynamique : l'interpréteur détermine le type à la volée lors de l'exécution du code. Dans l'exemple précédent, le type de `a` est `int`.

Le nom des variables ne doit pas commencer par un chiffre et ne doit pas comporter de tiret. Par contre, on peut utiliser « _ ». Exemple de variables :

```
v_exp=2    # v_exp est possible alors que v-exp ne peut pas représenter une
           # variable
b2=3       # la variable b2 prend la valeur 3. Par contre, la variable 2b
           # n'est pas autorisée
```

La fonction `print(a)` permet d'afficher la valeur de `a` :

```
print(a)
```

Cette instruction affiche la valeur de `a`.

Toutes les variables dans Python ont un type. La fonction `type(a)` retourne le type de la variable `a`.

```
print(type(a))  # affichage du type de la variable a
```

Le symbole # permet d'ajouter des commentaires. Les caractères après # font partie du commentaire dans le programme Python.

On obtient sur l'afficheur :

```
int
```

La variable `a` est de type `int`, c'est-à-dire entier.

```
b=5.2              # b est un nombre flottant dont la valeur est 5,2
print(type(b))     # affichage du type de b : float
```

**Attention**

On n'écrit pas `5,2` mais `5.2` avec Python.

On peut effectuer des calculs avec Python :

```
c=(a+b)/2     # calcul de la moyenne de a et b
d=3.1e-2      # 3,1e(-2)
```

La variable `b` est de type `float`, c'est-à-dire un nombre à virgule flottante.

Opérations de base sur les entiers (`int`) : `+, -, *, //, **, %`

```
n=28
n//10    # 2=quotient de la division euclidienne de n par 10
n%10     # 8=reste de la division euclidienne de n par 10
n**3     # n puissance 3
```

Opérations de base sur les flottants (`float`) : `+`, `-`, `*`, `/`, `**`

```
a=1/3     # a vaut 0.3333333333333333
2.6**(a)  # 2,6 puissance a
```

### Type `str` (chaîne de caractères)

Les chaînes de caractères (type `str`) sont des structures indicées immuables. On ne peut pas modifier les éléments d'une chaîne de caractères. On ajoute des apostrophes ou des guillemets autour d'un mot.

```
mot1= "C'est "   # mot1 est une chaîne de caractères de type str
n=len(mot1)      # n=6 (nombre de caractères)
mot2= 'un mot'   # on aurait pu écrire "un mot"
mot3=mot1+mot2   # concaténation de chaînes. On obtient "C'est un mot"
print(mot3[0])   # on obtient la première lettre de mot3
```

On peut afficher plusieurs éléments sur même ligne. Il suffit de les séparer par une virgule dans la fonction `print()`.

```
print("La moyenne de a et b est :",c)
```

On peut concaténer des chaînes de caractères :

```
mot4=mot1+mot2
print(mot4)
```

On ne peut pas additionner un entier et une chaîne de caractères.

```
d=a+mot1
```

On obtient le message d'erreur :

```
TypeError: can only concatenate str (not "int") to str
```

On peut convertir une chaîne de caractères en réel ou en entier.

```
C1='123'     # C1 est chaîne de caractères de type str
a=int(C1)    # conversion de C1 en entier
C2='12.5'    # C2 est une chaîne de caractères
b=float(C2)  # conversion de C2 en float
c=a**3+b**2  # calcul de a*a*a + b*b
print(c)     # affichage de c
```

On peut concaténer une chaîne de caractères *n* fois avec elle-même :

```
C1='ab'
C2=C1*3     # retourne 'abcabcabc'. Répétition de 'ab'
            # C1 est concaténée 3 fois avec elle-même
```

### Type `bool` (booléen)

On peut définir une variable qui vaut `True` (vrai) ou `False` (faux).

```
rep1=True   # rep1 vaut True. Le type est bool (booléen)
rep2=False  # rep2 vaut False. Le type est bool (booléen)
a=12
b=10
```

```
rep3=(a==12)and(b==20)  # False pour le test : a=12 et b=20
rep4=not (a==13)        # rep4=True, on pourrait écrire : rep4 = a!=13
rep5=(a==12)or(b==20)   # True pour le test : a=12 ou b=20
```

### Type `list` (liste)

Une liste est une collection ordonnée d'éléments. On utilise les crochets pour définir des listes avec Python. On peut avoir des éléments de type différent dans une liste. Pour créer une liste contenant les éléments 3, 5 et 8, on utilise l'instruction suivante :

```
L1=[3, 5, 8]
```

Création d'une liste par compréhension :

```
L2=[i**2 for i in range(6)]   # on obtient L2= [0, 1, 4, 9, 16, 25]
```

On peut concaténer deux listes en utilisant « + » :

```
L1=L1+[6,'mot']               # concaténation de la liste L1 et de la liste [6,'mot']
print(L1)                     # on obtient : [3, 5, 8, 6, 'mot']
```

On peut utiliser également la fonction `append()`.

```
L1.append(9)                  # on ajoute l'élément 9 à la liste L1
print(L1)                     # on obtient : [3, 5, 8, 6, 'mot', 9]
```

On peut supprimer le dernier élément ajouté dans la liste et récupérer sa valeur :

```
x=L1.pop()                    # x prend la valeur 9 de type int
print(L1)                     # on obtient : [3, 5, 8, 6, 'mot']
```

Pour créer une liste vide :

```
L3=[]                         # création d'une liste vide
L3.append([3, 2])             # la liste L3 vaut : [3, 2]
```

La fonction `len()` permet d'obtenir le nombre d'éléments dans une liste :

```
n=len(L1)                     # la variable n vaut 4
```

On peut concaténer une liste *n* fois avec elle-même :

```
L4=[3,2]*3                    # retourne [3, 2, 3, 2, 3, 2]. Répétition de [3, 2]
                              # la liste [3, 2] est concaténée 3 fois avec elle-même
```

On peut concaténer une liste avec une autre liste :

```
L5=L4+[1, 9]                  # on obtient : [3, 2, 3, 2, 3, 2, 1, 9]
```

**Attention**

« * » ne fait pas la multiplication par 4 de chaque élément de cette liste.

La fonction `sort()` serait rappelée dans un problème de concours.

```
L5.sort()      # trie la liste L5 en place, dans l'ordre croissant de ses éléments
               # cette fonction ne retourne rien
print(L5)      # [1, 2, 2, 2, 3, 3, 3, 9]
```

On verra dans le chapitre 10 différents algorithmes pour trier les listes.

### Indices des listes avec Python

On définit une liste `L5` :

```
L5=[-3,8.2,5,19]
n=len(L5)    # la longueur de la liste vaut n = 4
```

**Remarque**

Les indices des listes contenant *n* éléments sont numérotés de 0 à `n`–1 dans Python, contrairement à Matlab et Scilab, où les listes sont numérotées de 1 à `n`.

Pour obtenir le premier élément de la liste :

```
a=L5[0]     # la variable a prend la valeur -3
```

Pour obtenir le troisième élément de la liste `L5` :

```
b=L5[2]     # b vaut 5
```

Attention, le troisième élément de la liste `L5` a pour indice 2.

On a deux possibilités pour obtenir le dernier élément de la liste `L5` :

```
n=len(L5)
c=L5[n-1]   # valeur de l'élément d'indice n-1, c'est-à-dire
            # le dernier élément de L5
```

On peut modifier un élément d'une liste :

```
L5[1]=25                  # on modifie la valeur de l'élément d'indice 1
print('Liste L5 :',L5)    # sur l'afficheur, on a :
                          # "Liste L5 : [-3, 25, 5, 19]"
```

### Listes de listes

On peut représenter une matrice 2×3 par une liste `L` contenant 2 listes de longueur 3. Chacune de ces listes de longueur 3 représente une ligne de la matrice $\begin{pmatrix} 3 & 2 & 1 \\ 8 & 6 & 4 \end{pmatrix}$.

```
L=[[3,2,1], [8,6,4]]
```

Chaque élément de la liste est une liste.

Pour extraire le premier élément de la liste :

```
M=L[0]    # M vaut [3,2,1]
```

Pour récupérer le deuxième élément de `M` :

```
a=M[1]    # a vaut 2
```

On peut également écrire :

```
b=L[0][1] # b vaut 2
```

### Type `tuple` (**tuple**)

Les tuples sont des structures indicées immuables. Un tuple ressemble à une liste mais les éléments ne peuvent pas être modifiés une fois qu'ils sont créés. On utilise « `()` » pour définir un tuple alors qu'on utilise « `[]` » pour définir une liste.

```
L=(2, 5, 2)  # création d'un tuple contenant trois éléments
a=L[0]       # a vaut la valeur du premier élément du tuple
n=len(L)     # n vaut le nombre d'éléments du tuple L
```

On peut récupérer les valeurs des éléments du tuple. On peut modifier la valeur d'un élément d'une liste mais on ne peut pas le faire pour un tuple.

```
L[1]=8    # message d'erreur :
          # "TypeError: 'tuple' object does not support item assignment"
```

Python renvoie un message d'erreur indiquant que l'on ne peut pas modifier la valeur de `L[1]`.

Sauf indication contraire, on n'utilisera pas les tuples dans les programmes. Par contre, on verra que les fonctions renvoient des tuples et on cherchera à récupérer les différents éléments du tuple (voir exercice 4.1 « Recherche du minimum dans une liste non triée » dans le chapitre 4 « Algorithmes, recherche dans une liste, recherche par dichotomie »).

On peut définir un tuple en omettant les parenthèses :

```
L2=3, 6, 1         # on définit un tuple
print(type(L2))    # le type est tuple
L3=L+L2            # concaténation de deux tuples : L3=(2, 5, 2, 3, 6, 1)
L4=(2, 1)*3        # création avec répétition : L4=(2, 1, 2, 1, 2, 1)
```

### Type `deque` (deque)

Une deque (se prononce « dèque ») permet d'ajouter et de supprimer très rapidement des éléments aux extrémités droite et gauche (voir exercice 6.4 « Utilisation des deques » dans le chapitre 6 « Pile, file, deque, dictionnaire »). Pas d'extraction de tranche (ou slicing) avec les deques.

```
from collections import deque  : module permettant d'utiliser les deques
D=deque()            # création d'une deque vide D
D.append(8)          # ajoute 8 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
D1=deque([3, 8, 5])
for elt in D1:       # affichage de tous les éléments de la deque D1
    print(elt)
```

### Type `dict` (dictionnaire)

Voir exercice 6.5 « Comptage des éléments d'une liste à l'aide d'un dictionnaire » dans le chapitre 6 pour l'utilisation des dictionnaires.

## Extraction de tranche ou slicing pour les listes, tuples et chaînes de caractères

Lorsqu'on veut extraire des éléments d'une liste, d'un tuple ou d'une chaîne de caractères, on utilise la technique du slicing ou extraction de tranche. Il suffit de mettre entre crochets les indices correspondant au début et à la fin de la « tranche » et de les séparer par « : ». On utilise la syntaxe :

```
A[start:stop]
```

- `start` : indice de depart, inclus.
- `stop` : indice final, exclu.
- `stop – start` : nombre d'éléments de la liste `A` que l'on souhaite extraire (avec un pas de 1 par défaut).

### Liste

```
A=[3, 5, 1, 8, 10]     # liste de 5 éléments
```

Pour extraire les éléments [5, 1, 8] :

```
A[1:4]     # renvoie [5, 1, 8]
```

L'indice de départ vaut 1 avec `A[1]` = 5.

L'indice final vaut 4 – 1 = 3 avec `A[3]` = 8.

On récupère bien 3 éléments : [5, 1, 8].

Pour avoir un slicing avec un pas différent de 1, on utilise la syntaxe suivante :

```
A[start:stop:step]
```

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

On peut omettre `start`, `stop` ou `step`.

```
L=[2, 5, -3, 9, -4, 3]   # définition d'une liste
L1=L[:3]   # L1 contient les éléments d'indice i<3 : [2, 5, -3]
L2=L[3:]   # L2 contient les éléments d'indice i>=3 : [9, -4, 3]
L3=L[:]    # L3 contient tous les éléments de L
```

### Tuple, chaîne de caractères

On utilise la même technique pour les tuples et les chaînes de caractères.

```
L=(3, 5, 8, -2)   # tuple
L2=L[0:3]         # L2=(3, 5, 8)
c="C'est un mot"  # chaîne de caractères
c2=c[0:3]         # c2="C'e"
```

## Permutation de deux variables

On a très souvent besoin dans les programmes de permuter deux éléments.

On peut écrire :

```
x=3
y=5
c=x
x=y    # x vaut 5
y=c    # y vaut 3
```

On peut utiliser l'instruction `u,v = v,u` qui permute `u` et `v` :

```
x=3
y=5
x,y = y,x    # on obtient x=5 et y=3
```

On peut réduire le programme à deux lignes :

```
x,y = 3,5     # x=3 et y=5
x,y = y,x     # on obtient x=5 et y=3
```

La première ligne définit un tuple qui vaut (3,5).

La deuxième ligne définit un tuple en fonction d'un autre tuple, d'où la permutation des deux éléments.

# Opérateurs arithmétiques

| | |
|---|---|
| `+` | addition |
| `-` | soustraction |
| `*` | multiplication |
| `/` | division |
| `//` | quotient de la division euclidienne |
| `%` | reste de la division euclidienne |
| `**` | exponentiation ou puissance |
| `abs()` | valeur absolue |
| `round(x)` | renvoie la valeur de l'entier le plus proche de *x*. Si deux entiers sont équidistants, l'arrondi se fait vers la valeur paire |
| `round(x,2)` | arrondit *x* avec une précision de 2 chiffres après la virgule |

```
x=2**4            # x vaut 16=2*2*2*2
q=9//2            # quotient de la division euclidienne de 9 par 2
r=9%2             # reste de la division euclidienne de 9 par 2
a=abs(-9.2)       # a vaut 9.2
b=round(8.5163,2) # arrondi avec 2 chiffres après la virgule : 8.52
c=(a+b)/x         # calcul avec des parenthèses
```

# Tests

On utilise la syntaxe **if** (« si », en français) pour faire un test. Les opérateurs de comparaison sont :

| | |
|---|---|
| `==` | égal à |
| `!=` | différent de |
| `<` | strictement inférieur |
| `<=` | inférieur ou égal |
| `>` | strictement supérieur |
| `>=` | supérieur ou égal |

Il faut ajouter « `:` » à la fin de la ligne.

Les lignes devant être exécutées si la condition est vérifiée doivent toutes être indentées.

```
L2=[18.2, 9.5, 15]
note=L2[1]
if note==10:              # ne pas oublier ":" à la fin de la ligne
    print("Note = 10")    # indentation pour exécuter cette ligne si note=10
```

On crée une liste `L2` contenant 4 éléments. On récupère la deuxième valeur de la liste `L2` dans la variable note. On teste si cette note vaut 10, alors on obtient sur l'afficheur :

```
Note = 10
```

On peut utiliser la syntaxe `if…else`.

```
if note==10:              # ne pas oublier ":" à la fin de la ligne
    print("Note = 10")    # indentation pour exécuter cette ligne si note=10
else:    # ne pas oublier ":" à la fin de la ligne
    print('Note différente de 10')
# indentation pour exécuter cette ligne si note est différent de 10
```

On peut chercher à effectuer plusieurs tests si le premier n'est pas vérifié. Dans ce cas, on utilise la syntaxe `elif` qui signifie « sinon si ». Si le test précédent n'est pas vérifié, il effectue un nouveau test et ainsi de suite.

```
if note==10:              # ne pas oublier ":" à la fin de la ligne
    print("Note = 10")    # indentation pour exécuter cette ligne si note=10
elif note <10:
    print('Note strictement inférieure à 10')
    print(note)           # affichage sur une autre ligne
elif note<=15:
    print('Note inférieure à 15 :',note)
else:
    print('Note strictement supérieure à 15')
```

Les tests sont effectués dans l'ordre du programme. Si une condition est vérifiée, les tests ultérieurs avec `elif` ne sont pas effectués.

On peut avoir des opérateurs logiques dans les tests.

```
a==b         # renvoie un booléen True si a=b et False sinon
rep==True    # renvoie un booléen True si rep=True et False sinon
```

Si on veut réunir des expressions booléennes, on peut utiliser les opérateurs : `and` (intersection), `or` (union), `not` (négation).

```
a==a         # renvoie un booléen True
not(a==a)    # renvoie un booléen False
```

**Remarque**

On peut utiliser `&` à la place de `and`.

```
a, b=3, 2    # affectation sur une seule lignes des variables a et b
rep=False    # la variable rep est un booléen
if (a==b) and (rep==True):    # teste si a=b et si rep=True
    print(a)
else:
    print(a+b)
```

# Importation de modules

Des fonctions traitant d'un même domaine sont regroupées dans des modules (par exemple les fonctions mathématiques `cos`, `sin`, `tan`… sont regroupées dans le module `math`). Différents modules peuvent être regroupés dans une bibliothèque. On utilise l'instruction `import module` pour importer un module.

### Module math

```
import math  # importation du module math
dir(math)    # liste des fonctions du module math
```

Le module `math` contient des fonctions et des variables : `cos()`, `sin()`, `tan()`, `exp()`, `sqrt()` (racine carrée), `log()` (logarithme népérien), `log10()` (logarithme décimal), `pi` (nombre π)…

Pour avoir plus d'informations sur une fonction ou une variable :

```
?math.pi     # Type : float. String form : 3.141592653589793
?math.cos    # Return the cosine of x (measured in radians)
```

Pour utiliser les fonctions et les variables du module `math` :

```
a=math.pi/4
b=math.cos(math.pi/4)
c=math.sin(math.pi)
print(b)     # affiche 0.7071067811865476
print(c)     # affiche 1.2246467991473532e-16
```

On peut être surpris que Python n'affiche pas 0 lors du calcul de sin(π). Le type `float` ne permet pas le calcul exact mais une valeur approchée avec une précision d'environ $10^{-16}$ (représentation des flottants sur des mots de taille fixe).

Pour effectuer le test `sin(x)==0`, on n'utilisera pas l'instruction :

```
m.sin(x)==0          # retourne False alors que x=pi
```

mais les instructions suivantes :

```
eps=1**-8            # on choisit une valeur pour eps
abs(m.sin(x))<eps    # retourne True si x=pi
```

Lorsqu'on utilise l'instruction `from math import *`, il n'est plus nécessaire d'ajouter le nom du module pour utiliser ses fonctions :

```
from math import *   # module math
a=pi/4
```

Certaines fonctions portent le même nom dans des bibliothèques différentes. Il est donc préférable de ne pas utiliser `from math import *` mais plutôt `import math`. On peut renommer le module `math` en `m` par exemple :

```
import math as m     # module math renommé m
a=m.pi/4
```

On peut importer des fonctions et des variables d'un module :

```
from math import cos, sin, tan, pi
a=cos(pi/4)
```

### Module random

Les fonctions du module `random` seraient rappelées dans un problème de concours.

```
import random as rd  # module random renommé rd
n=20
i=rd.randint(0, n)   # indice aléatoire i compris entre 0 inclus et n-1 inclus
M=rd.random()        # nombre flottant aléatoire M tel que 0<=M<1
```

# Importation de bibliothèques

La bibliothèque `matplotlib` regroupe plusieurs modules concernant les graphiques. L'instruction suivante permet d'importer le module `pyplot` de la bibliothèque `matplotlib` que l'on renomme `plt` :

```
import matplotlib.pyplot as plt  # module matplotlib.pyplot renommé plt
```

On peut avoir la liste des fonctions du module `pyplot` :

```
dir(plt)
```

L'instruction suivante permet de représenter `y` en fonction de `x` en utilisant la fonction `plot()` du module `pyplot` (voir chapitre 2 « Représentation graphique ») :

```
plt.plot(x, y)
```

On utilisera la bibliothèque `PIL` dans le chapitre 9 « Matrices de pixels et images » :

```
from PIL import Image            # module Image de la bibliothèque PIL
```

En physique, on utilise la bibliothèque `numpy`.

# Boucle `for`

Les boucles permettent de répéter des opérations un certain nombre de fois. On utilise l'instruction `for i in range()`.

```
for i in range(3): # i varie entre 0 inclus et 3 exclu avec un pas égal à 1
    b=i*2+3        # indentation pour exécuter cette ligne à chaque étape
    print(b)       # indentation pour exécuter cette ligne à chaque étape
```

`range(3)` signifie que la boucle `for` va être exécutée 3 fois.

Il faut bien indenter les lignes devant être exécutées à chaque étape de la boucle `for`.

Ne pas oublier d'écrire « `:` » à la fin de la ligne `for`.

*i* prend successivement les valeurs 0, 1 et 2 :

- Première étape de la boucle `for` : `i` = 0. On obtient « 3 » sur l'afficheur.
- Deuxième étape de la boucle `for` : `i` = 1. On obtient « 5 » sur l'afficheur.
- Troisième étape de la boucle `for` : `i` = 2. On obtient « 7 » sur l'afficheur.

On ne retrouve pas la syntaxe « `end` » comme dans d'autres langages. La fin de l'indentation représente la fin de la boucle `for`.

### Syntaxe de la fonction `range()`

```
for i in range(start, stop, step):
```

`i` est un entier qui varie de `start` inclus à `stop` exclu avec un pas égal à `step` :

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

```
for i in range(n):    # Python prend par défaut : start = 0 et step = 1
```

`i` varie entre 0 inclus et `n` exclu avec un pas égal à 1.

La boucle `for` sera donc exécutée `n` fois.

### Parcours des éléments d'une liste ou d'une chaîne de caractères

```
L=[3, 5, 8]      # liste L contenant 3 éléments
for elt in L:    # elt prend successivement la valeur d'un élément de L
    print(elt)   # affichage d'un élément de L à chaque étape
```

`elt` prend à chaque étape la valeur d'un élément de `L`.

La boucle `for` sera exécutée 3 fois.

## Boucle `while` (tant que)

Il faut bien indenter les lignes devant être exécutées à chaque étape de la boucle `while`.

```
i=0
while i!=3:    # ne pas oublier ":" à la fin de la ligne while
    i=i+1      # on incrémente i de 1 à chaque étape
    b=i*2+3    # b est calculé à chaque étape
    print(b)   # b est affiché à chaque étape
```

La variable `i` est initialisée à 0 et incrémentée de 1 à chaque étape de la boucle `while`. On l'appelle un compteur.

Si la variable `i` est incrémentée d'une valeur différente de 1 ou décrémentée, on l'appellera un accumulateur.

**Remarque**

On peut utiliser l'instruction suivante :

```
i+=1    # la variable i est incrémentée de 1
```

- La variable `i` est initialisée à 0.
- Première étape de la boucle `while` (`i` est différent de 3) : `i` est incrémenté de 1 et vaut 1. Python affiche « 5 ».
- Deuxième étape de la boucle `while` (`i` est différent de 3) : `i` est incrémenté de 1 et vaut 2. Python affiche « 7 ».
- Troisième étape de la boucle `while` (`i` est différent de 3) : `i` est incrémenté de 1 et vaut 3. Python affiche « 9 ».

À la fin de la troisième étape, `i` vaut 3. La condition « *i* différent de 3 » n'est plus vérifiée. Le programme quitte la boucle `while`.

L'instruction `break` fait sortir d'une boucle `while` ou `for` et passe à l'instruction suivante (voir exercice 10.8 « Tri à bulles » dans le chapitre 10 « Algorithmes de tri ». Lorsqu'il y a plusieurs boucles imbriquées, l'instruction `break` ne fait sortir que de la boucle la plus interne.

## Définition d'une fonction

On utilise l'instruction `def` pour définir une fonction dans Python. Dans la parenthèse, on indique les arguments d'entrée.

Le corps de la fonction est indenté d'un niveau (touche TAB ou 4 espaces). Les variables définies dans le corps de la fonction sont des variables locales.

Une fonction peut renvoyer des objets avec l'instruction `return`. Le programme quitte immédiatement la fonction quand il rencontre l'instruction `return`.

### Exemple de fonction

On souhaite définir la fonction $f(t) = 3\cos(5t)$.

```
from math import *  # module math
def f(t):           # argument d'entrée : t
    y=3*cos(5*t)    # calcul de y. Ne pas oublier le symbole *
    return y        # argument de sortie : y
```

On peut utiliser cette fonction dans le programme principal suivant. L'utilisateur tape au clavier la valeur de `t`. On obtient une chaîne de caractères que l'on convertit en `float`. Il reste à appeler la fonction pour obtenir le résultat dans la variable `res`.

```
rep=input("Taper la valeur de t :")     # rep est une chaîne de caractères
T=float(rep)  # conversion de rep en float
res=f(T)      # le résultat de la fonction est affecté dans la variable res
print("f("+rep+")=", res)
```

**Remarque**

On pourrait écrire également :

```
t=float(input("Taper la valeur de t :"))    # rep est une chaîne de caractères
print("f("+rep+")=",f(t))
```

Si on écrit `return val1, val2, val3`, on récupère un tuple de 3 éléments après l'appel de cette fonction. Voir exercice 4.1 « Recherche du minimum dans une liste non triée » pour l'utilisation de ces données.

## Objets muables, objets immuables

Il existe deux types d'objets dans Python :

- Les objets dont la valeur peut changer sont dits muables (ou *mutable* en anglais) : listes, dictionnaires, deques (voir chapitre 6 « Piles, files, deques, dictionnaire »)…
- Les objets dont la valeur ne peut pas changer sont dits immuables (ou *immutable* en anglais) : entiers, flottants, booléens, chaînes de caractères, tuples…

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

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

- Création d'un objet muable (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]`.

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 muable) : `[10, 2, 3, 4, 12]`.

## Copie superficielle, copie profonde

```
import copy        # module copy
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)  # L1=[12, 2, [30, 4], 5]
print('L2 =', L2)  # L2=[12, 2, [30, 4], 5]
print('L3 =', L3)  # L3=[1, 2, [30, 4], 5]
print('L4 =', L4)  # L4=[1, 2, [3, 4], 5]
```

- La fonction `copy()` du module `copy` réalise une copie superficielle. Les éléments sont copiés s'il n'y pas de structure imbriquée. Si les éléments sont des listes par exemple, alors l'adresse mémoire des listes est copiée.
- La fonction `deepcopy()` du module `copy` réalise une copie profonde pour les structures imbriquées. Si les éléments sont des listes, alors la copie profonde copie bien les listes imbriquées.

## Passage par valeur, passage par référence, variable globale, variable locale dans les fonctions

Une variable locale existe uniquement dans la table des variables d'une fonction. Elle est créée à l'appel de la fonction et détruite à la fin de la fonction.

Une variable globale est définie en dehors d'une fonction et peut être utilisée et modifiée dans la fonction. Il faut utiliser `global` pour déclarer une variable globale dans une fonction.

Lorsqu'une expression fait référence à une variable à l'intérieur d'une fonction (variable `d` par exemple), Python cherche la valeur définie à l'intérieur de la fonction et à défaut la valeur dans l'espace global du programme.

```
a=3
b=3
d=5
x=10
L=[3, 5, 8]   # définition de la liste L
print('Liste L',L)
def calcul(x, L):
    global a  # a est une variable globale
    x=2*x     # la valeur de x est multipliée par 2
    a=4       # la variable globale a est modifiée
              # en quittant la fonction calcul, on retrouve a = 4
    b=4       # cette variable est locale
    # en quittant la fonction, on ne retrouve pas cette valeur
    # car b=4 est défini localement dans la fonction calcul
    c=3       # cette variable est locale
    # cette variable existe uniquement dans la fonction calcul
    print(d)  # Python affiche 5
    L[0]=2
    # en quittant la fonction, les variables locales b et c sont détruites
    return a+b+c+x
print('Résultat de la fonction :',calcul(x, L))  # affiche 31
print(L) # la liste a été modifiée car L est passé par référence : [2, 5, 8]
print(a) # la variable globale a été modifiée : affiche 4
print(b) # b n'a pas été modifiée par la fonction calcul : affiche 3
print(x) # affiche 10
print(c) # erreur car c n'existe que dans la fonction calcul
```

Dans Python, les listes, les dictionnaires, les deques sont passés par référence et non par valeur.

L'argument d'entrée `L` de la fonction `calcul` fait référence à la liste `[3, 5, 8]` qui est un objet muable. La liste `L` est passée par référence. Lorsqu'on modifie `L` dans la fonction `f`, on modifie la liste `[3, 5, 8]`. La variable `L` dans la fonction `f` fait référence à la même adresse mémoire que la variable `L` dans le programme principal.

Il est donc inutile d'écrire `return(L)` à la fin de la fonction `calcul`. La fonction modifie la liste `L` et on retrouve la modification après avoir quitté la fonction.

On distingue deux cas importants pour les arguments d'entrée :

- Objet muable (liste, dictionnaire, deque…) : 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 immuable (entier, nombre flottant, booléen, chaîne de caractères, tuple…) : toute modification de cet objet dans la fonction n'est pas visible en dehors de la fonction. Pour simplifier, tout se passe comme si ces objets étaient passés par valeur.

# Types des arguments dans les fonctions

Certains sujets de concours (Centrale-Supélec en particulier) utilisent des annotations en précisant les types des arguments et du résultat des fonctions.

On considère une fonction `F1` qui admet pour arguments d'entrée un entier `n`, un nombre flottant `x`, une chaîne de caractères `c1` et une liste `L`. Elle retourne un entier, un réel et une liste.

```
def F1(n, x, c1, L):
    a=3
    b=3.2
    L2=[a+b, 1]
    return a, b, L2
```

On peut détailler dans Python les types des arguments d'entrée et du résultat de la fonction :

```
def F1(n :int, x :float, c1 :str, L :list) -> (int, float, list):
    a=3
    b=3.2
    L2=[a+b,1]
    return a, b, L2

res=F1(3, 2, 'mot', [3,2])       # on récupère un tuple de 3 éléments
print(res[2])                    # affiche [6.2, 1] le troisième élément du tuple
```

On considère une fonction `F2` qui admet pour arguments une liste L. Cette fonction ne retourne rien.

```
def F2(L):
    a=4
    b=3.2
    L[0]=a+b
```

On peut détailler dans Python les types des arguments d'entrée et du résultat de la fonction :

```
def F2(L:list) -> None:
    # L est passé par référence
    a=4           # variable locale
    b=3.2         # variable locale
    L[0]=a+b
    # cette fonction ne retourne rien

L1=[3, 2]
F2(L1)
print(L1)         # on obtient [7.2, 2] et non [3, 2]
# L1 a bien été modifiée par F2 car L1 est passée par référence

res=F2(L1)        # la fonction ne retourne rien
print(res)        # res prend la valeur None
print(type(res))  # le type de res est 'NoneType'
```

La liste `L1` est passée par référence. La fonction `F2` modifie `L1` alors qu'elle ne retourne rien.

La variable `a` prend la valeur `None`, c'est la valeur retournée par une fonction qui ne retourne rien.

Le type de `a` est `NoneType`.

## Commande Python sur plusieurs lignes

Pour la clarté du programme, on peut écrire une commande Python sur plusieurs lignes. On utilise « \ » en fin de ligne.

L'exemple suivant permet de calculer *y* en utilisant trois lignes :

```
import math
y=(math.exp(-math.pi)+3+math.pi)\
/(5*math.sin(math.pi/4)+2)\
+math.log(2)

print(y)  # affiche le résultat : 1.81
```

## Commentaires dans les programmes Python

Les commentaires commencent par « # ». Tous les caractères situés sur la ligne après « # » ne seront pas pris en compte lors de l'exécution du programme Python.

Exemple :

```
print(y)  # affiche le résultat : 1.81
```

## Assertion

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

```
def calcul(L):      # argument d'entrée : liste L
    # la fonction retourne la somme du premier élément
    # et du dernier élément de la liste L
    # on pourrait écrire : def fonction(L:[float])->float:
    # L est une liste de nombres flottants
    assert type(L)==list
    assert len(L)>1
    a=L[0]+L[len(L)-1]
    return a        # argument de sortie : a
```

La fonction `calcul` peut être appelée si le type de `L` est `list`.

Le programme teste si `assert 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`.

La fonction `calcul` peut être appelée si le nombre d'éléments de `L` est strictement supérieur à 1. Le programme teste si `len(L)>1`. Si la condition est vérifiée, le programme continue à s'exécuter normalement. Si la condition `len(L)>1` 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)>=1 AssertionError`.

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

# Énoncés des exercices

## 1.1

■□□□

**Moyenne et écart-type de notes**

**a)** Proposer une fonction `moyenne` qui admet comme argument d'entrée une liste non vide de notes. Cette fonction retourne la moyenne des notes.

**b)** Proposer une fonction `ecart_type` qui admet comme argument d'entrée une liste non vide de notes. Cette fonction retourne l'écart-type des notes.

**c)** Les entêtes des fonctions peuvent être annotés pour préciser les types des paramètres et du résultat. Ainsi,

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

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, `c` une chaîne de caractères et le type de `u` n'est pas précisé. Cette fonction renvoie une liste.

Écrire une fonction `moy` qui admet comme argument `L` une liste non vide de nombres réels ou flottants et retourne la moyenne de la liste `L`. Annoter l'entête 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`.

## 1.2

■■□□

**Fonction factorielle**

Proposer une fonction itérative `fact` qui admet comme argument un entier naturel $n$ et retourne la factorielle de $n$.

## 1.3

■■■□

**Calcul des termes d'une suite**

On considère $(u_n)_{n\in\mathbb{N}}$ la suite des nombres définie par :

$u_0 = 1$, $\forall n \in \mathbb{N}$, $u_{n+1} = 3u_n + 5$.

**a)** Proposer une fonction qui admet comme argument un entier naturel $n$ et retourne $u_n$. On n'utilisera pas de liste dans la fonction.

**b)** Proposer une fonction qui admet comme argument un entier naturel $n$ et retourne la liste $L = [u_0, u_1, u_2, \ldots, u_n]$.

## 1.4

■■■■

**File de voitures et codage binaire (Mines Ponts 2017)**

On considère un ensemble de voitures sur une file à sens unique. Une file `L` de longueur $n$ est représentée par $n$ cases. Une case peut contenir au plus une voiture. Lorsqu'une case d'indice `i` de la file est occupée par une voiture, `L[i]` vaut `True`, sinon `L[i]` vaut `False`. Afin de comparer plus efficacement les files

représentées par des listes de booléens, on remarque que ces listes représentent un codage binaire où `True` correspond à 1 et `False` à 0.

**a)** Soit `L` une liste représentant une file de longueur $n$ et $i$ un entier tel que $0 \leq i < n$. Définir en Python la fonction `occupe(L, i)` qui renvoie `True` lorsque la case d'indice $i$ de la file est occupée par une voiture et `False` sinon.

**b)** Combien existe-t-il de files différentes de longueur $n$ ? Justifier votre réponse.

**c)** Écrire la fonction `versEntier(L)` prenant une liste de booléens en paramètre et renvoyant l'entier correspondant. Par exemple, l'appel `versEntier([True, False, False])` renverra 4.

**d)** On veut écrire la fonction inverse de `versEntier`, transformant un entier en une liste de booléens. Que doit être au minimum la valeur de `taille` pour que le codage obtenu soit satisfaisant ? On suppose que la valeur de `taille` est suffisante. Quelle condition booléenne faut-il écrire en quatrième ligne du code ci-dessous ?

```
def versFile(n, taille):
    res=taille*[False]
    i=taille-1
    while ...:
        if (n%2) !=0:    # % est le reste de la division euclidienne de n par 2
            res[i]=True
        n = n//2         # // est le quotient de la division euclidienne de n par 2
        i = i - 1
    return res
```

## 1.5

**Recherche d'un indice dans une liste (Mines Ponts 2018)**

On s'intéresse à des mesures de niveau de la surface libre de la mer. On appelle *horodate* un ensemble (fini) des mesures réalisées sur une période de 20 minutes à une fréquence d'échantillonnage de 2 Hz. Les informations de niveau de surface libre (déplacement vertical en m) sont stockées dans une liste de flottants `liste_niveaux`. On suppose qu'aucun des éléments de cette liste n'est égal à la moyenne.

**a)** Proposer une fonction `moyenne` prenant en argument une liste non vide `liste_niveaux` et retournant sa valeur moyenne.

**b)** Proposer une fonction `ind_premier_pzd(liste_niveaux)` retournant, s'il existe, l'indice du premier élément de la liste tel que cet élément soit supérieur ou égal à la moyenne et l'élément suivant soit inférieur ou égal à la moyenne. Cette fonction devra retourner « −1 » si aucun élément vérifiant cette condition n'existe.

**c)** Proposer une fonction retournant l'indice $i$ du dernier élément de la liste tel que cet élément soit supérieur ou égal à la moyenne et l'élément suivant soit inférieur ou égal à la moyenne. Cette fonction devra retourner « −2 » si aucun élément vérifiant cette condition n'existe. On cherchera à proposer une fonction de complexité en $O(1)$ dans le meilleur des cas.

## 1.6

**Analyse de vagues (Mines Ponts 2018)**

On s'intéresse à des mesures de niveau de la surface libre de la mer. On appelle *horodate* un ensemble (fini) des mesures réalisées sur une période de 20 minutes à une fréquence d'échantillonnage de 2 Hz. Les

informations de niveau de surface libre (déplacement vertical en m) sont stockées dans une liste de flottants `liste_niveaux`. On suppose qu'aucun des éléments de cette liste n'est égal à la moyenne. On considère que la mesure de houle est représentée par un signal $\eta(t) \in \mathbb{R}, t \in [0, T]$. On appelle niveau moyen $m$ la moyenne de $\eta(t)$ sur [0, $T$]. On définit $Z_1$, $Z_2$, …, $Z_n$ l'ensemble (supposé fini) des Passages par le Niveau moyen en Descente (PND) (voir figure 1). À chaque PND, le signal traverse la valeur $m$ en descente. On suppose $\eta(0) > m$ et $\frac{d\eta}{dt}(0) > 0$. On en déduit que $\eta(t) - m \geq 0$ sur $[0, Z_1]$. Les hauteurs de vague $H_i$ sont définies par les différences :

$$\begin{cases} H_1 = \max\limits_{t \in [0, Z_1]} \eta(t) - \min\limits_{t \in [Z_1, Z_2]} \eta(t) \\ H_i = \max\limits_{t \in [Z_{i-1}, Z_i]} \eta(t) - \min\limits_{t \in [Z_i, Z_{i+1}]} \eta(t) \text{ pour } 2 \leq i < n \end{cases}$$

On définit les *périodes de vagues* par $T_i = Z_{i+1} - Z_i$. La fréquence d'échantillonnage est égale à 2 Hz.

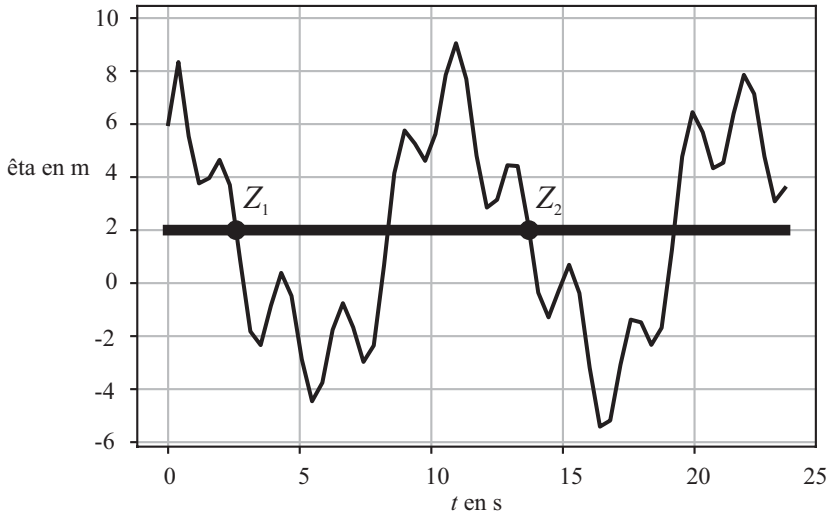



**Figure 1** Passages par le Niveau moyen en Descente (PND). Ici, la moyenne m vaut 2.

On souhaite stocker, dans une liste `successeurs`, les indices des points succédant (strictement) aux PND (voir figure 2).

**a)** On propose la fonction `construction_successeurs`, qui retourne la liste `successeurs`. Compléter les lignes 6 et 7.

```
1  def construction_successeurs(liste_niveaux):
2      n=len(liste_niveaux)
3      successeurs=[]
4      m=moyenne(liste_niveaux)
5      for i in range(n-1):
6          if              # À compléter
7                          # À compléter
8      return successeurs
```

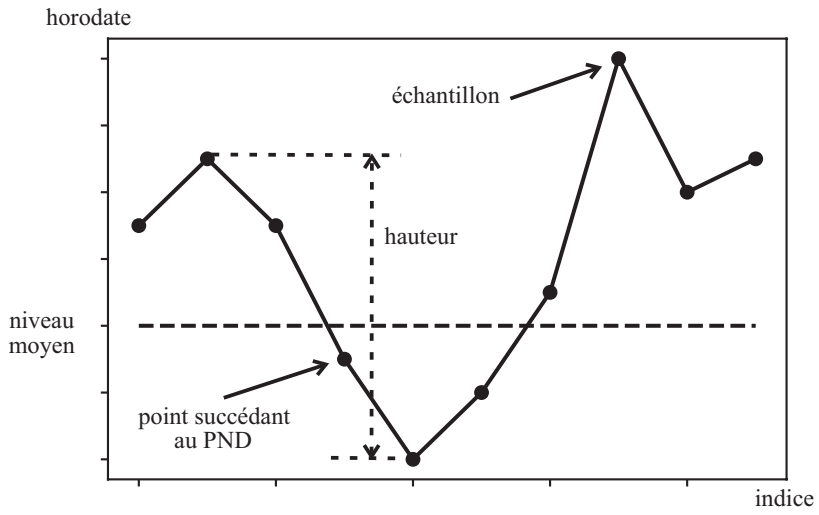



**Figure 2** Propriétés d'une vague

**b)** Proposer une fonction `decompose_vagues(liste_niveaux)` qui permet de décomposer une liste de niveaux en liste de vagues. On omettra les données précédant le premier PND et celles succédant au dernier PND. Ainsi `decompose_vagues([1, -1, -2, 2, -2, -1, 6, 4, -2, -5])` (noter que cette liste est de moyenne nulle) retournera « [–1, –2, 2],[–2, –1, 6, 4] ».

**c)** On désire maintenant caractériser les vagues. Ainsi, on cherche à concevoir une fonction `proprietes(liste_niveaux)` retournant une liste de listes à deux éléments `[Hi, Ti]` permettant de caractériser *chacune des vagues i* par ses attributs :

- `Hi`, sa hauteur en mètres (m) (voir figure 2) ;
- `Ti`, sa période en secondes (s).

Proposer une fonction `proprietes (liste_niveaux)` réalisant cet objectif. On pourra utiliser les fonctions de Python `max(L)` et `min(L)` qui retournent le maximum et le minimum d'une liste `L`, respectivement.

**d)** Plusieurs indicateurs sont couramment considérés pour définir l'état de la mer. Parmi eux, on note :

- $H_{\max}$ : la hauteur de la plus grande vague observée sur l'intervalle d'enregistrement [0, $T$] ;
- $H_{1/3}$ : la valeur moyenne des hauteurs du tiers supérieur des plus grandes vagues observées sur [0, $T$] ;
- $T_{H\,1/3}$ : la valeur moyenne des périodes du tiers supérieur des plus grandes vagues observées [0, $T$].

Proposer une fonction prenant en argument la liste `liste_niveaux` et retournant $H_{\max}$.

## 1.7

### Loi de Poisson et loi binomiale

Soient $n$ un entier naturel strictement positif et $p$ un réel compris entre 0 et 1. On considère $X$ et $Y$ deux variables aléatoires à valeurs dans $\mathbb{N}$ sur un espace probabilisé donné. $X$ suit une loi de Poisson de paramètre $\lambda = np$ et $Y$ suit une loi binomiale de paramètres $(n, p)$.

**a)** Proposer une fonction `Px`, d'arguments $k$, $n$ et $p$, renvoyant la valeur de :

$$P(X = k) = \frac{\lambda^k}{k!}\exp(-\lambda)$$

$k!$ (factorielle de $k$) s'obtient par `math.factorial(k)` du module `math`.

Écrire le programme principal permettant d'afficher la liste des valeurs de $P(X = k)$ pour $k \in \mathbb{N}, 0 \leq k \leq 30$, avec $n = 30$ et $p = 0{,}1$.

**b)** Proposer une fonction `Py`, d'arguments $k$, $n$ et $p$, renvoyant la valeur de :

$$P(Y = k) = \binom{n}{k} p^k (1-p)^{n-k} = \frac{n!}{k!(n-k)!} p^k (1-p)^{n-k}$$

Écrire le programme principal permettant d'afficher la liste des valeurs de $P(Y = k)$ pour $k \in \mathbb{N}, 0 \leq k \leq 30$, avec $n = 30$ et $p = 0{,}1$.

**c)** Soit $k \in \mathbb{N}$. On rappelle que, sous certaines conditions sur $n$ et $p$, la probabilité $P(Y = k)$ peut être approchée par $P(X = k)$. Proposer une fonction `Ecart` d'arguments $n$ et $p$, renvoyant le plus grand des nombres $\left|P(Y = k) - P(X = k)\right|$, pour $0 \leq k \leq n$.

**d)** Soit $e$ un réel strictement positif. Proposer une fonction `N`, d'arguments $e$ et $p$, renvoyant le plus petit $0 \leq k \leq n$ entier $n$ tel que `Ecart(n, p)` soit inférieur ou égal à $e$.

**e)** Écrire le programme principal permettant d'afficher le résultat de la fonction `N` dans les deux cas suivants :

- $e = 0{,}008$ ; $p = 0{,}075$ ;
- $e = 0{,}005$ ; $p = 0{,}075$.

## 1.8

**Nombres premiers (Mines Ponts 2019)**

Le crible d'Ératosthène est un algorithme qui permet de déterminer la liste des nombres premiers appartenant à l'intervalle $[\![1, n]\!]$. Son pseudo-code s'écrit comme suit (algorithme 1 : crible d'Ératosthène) :

**Données** : $N$, entier supérieur ou égal à 1

**Résultat** : *liste_bool*, liste de booléens

début

liste_bool ← liste de N booléens initialisés à Vrai ;

Marquer comme Faux le premier élément de *liste_bool* ;

**pour** *entier* $i \leftarrow 2$ à $\left\lfloor \sqrt{N} \right\rfloor$ **faire**

**si** i n'est pas marqué comme Faux dans liste_bool **alors**

Marquer comme Faux tous les multiples de $i$ différents de $i$ dans *liste_bool* ;

fin

fin

**retourner** *liste_bool*

fin

À la fin de l'exécution, si un élément de *liste_bool* vaut Vrai alors le nombre codé par l'indice considéré est premier. Par exemple pour N = 4, une implémentation Python du crible renvoie [False True True False].

**a)** Sachant que le langage Python traite les listes de booléens comme une liste d'éléments de 32 bits, quelle est (approximativement) la valeur maximale de N pour laquelle *liste_bool* est stockable dans une mémoire vive de 4 Go ?

**b)** Quel facteur peut-on gagner sur la valeur maximale de N en utilisant une bibliothèque permettant de coder les booléens non pas sur 32 bits mais dans le plus petit espace mémoire possible pour ce type de données (on demande de le préciser) ?

**c)** Écrire la fonction `erato_iter(N)` qui implémente l'algorithme 1 pour un paramètre N qui est un entier supérieur ou égal à 1.

L'approche systématique qui précède est inefficace car elle revient à attendre d'avoir généré la liste de tous les nombres premiers inférieurs à une certaine valeur pour en choisir ensuite quelques-uns au hasard. Une meilleure idée est d'utiliser des tests probabilistes de primalité. Ces tests ne garantissent pas vraiment qu'un nombre est premier. Cependant, au sens probabiliste, si un nombre réussit un de ces tests alors la probabilité qu'il ne soit pas premier est prouvée être inférieure à un seuil calculable. En suivant cette idée, une nouvelle approche est la suivante :

1) Générer un entier pseudo-aléatoire (voir ci-dessous).
2) Vérifier si cet entier a de fortes chances d'être premier.
3) Recommencer tant que le résultat n'est pas satisfaisant.

Pour générer un entier pseudo-aléatoire $A$, on se base sur un certain nombre d'itérations de l'algorithme Blum Blum Shub, décrit comme suit. On initialise $A$ à zéro au début de l'algorithme et pour chaque itération ($i \geq 1$) on calcule :

$$x_i = \text{reste de la division euclidienne de } x_{i-1}^2 \text{ par M} \qquad (1)$$

où $M$ est le produit de deux nombres premiers quelconques et $x_0$ une valeur initiale nommée « graine » choisie aléatoirement. On utilise ici l'horloge de l'ordinateur comme source pour $x_0$. Puis, pour chaque $x_i$, s'il est impair, on additionne $2^i$ à $A$.

**d)** On répète (1) pour $i$ parcourant $[\![1, N-1]\!]$. Quelle sera la valeur de $A$ si $x_i$ est impair à chaque itération ?

**e)** Compléter (avec le nombre de lignes que vous jugerez nécessaire) la fonction `bbs(N)` donnée ci-dessous qui réalise ces itérations. La graine est un entier représentant la fraction de seconde du temps courant, par exemple 1528287738.7931523 donne la graine 7931523. Le paramètre `N` est un entier non nul.

```
1  ...
2  ...
3  def bbs (N) :
4      p1 = 24375763
5      p2 = 28972763
6      M= p1 * p2
7      # calculer la graine
8      … (à compléter)
9      A = 0
10     for i in range(N) :
11         if  … (à compléter)  # si xi est impair
12             A = A + 2 ** i
13         # calculer le nouvel xi
14         xi = … (à compléter)
15     return (A)
```

On rappelle que la fonction `time()` du module `time` renvoie un flottant représentant le nombre de secondes depuis le 01/01/1970 avec une résolution de $10^{-7}$ seconde (horloge de l'ordinateur).

# Du mal à démarrer ?

## 1.1

**a)** Initialiser une variable `som` et écrire une boucle `for` pour décrire tous les éléments de la liste.

**b)** Appeler la fonction `moyenne` dans la fonction `ecart_type`.

**c)** Dans la fonction `moy`, ajouter une ligne `assert type(L)==list.`

## 1.2

Définir une variable `res`. Multiplier `res` par un entier à chaque étape de la boucle `for`.

## 1.3

**a)** Définir une variable `res`. Multiplier `res` par un entier à chaque étape de la boucle `for`.

**b)** Définir une liste. Ajouter un élément dans la liste à chaque étape de la boucle `for`.

## 1.4

**a)** La case d'indice *i* contient la valeur à retourner.

**b)** Définir une boucle en commençant par le dernier élément de la liste.

**c)** Envisager une exécution pas à pas du programme.

**d)** Envisager une exécution pas à pas du programme.

## 1.5

**a)** Initialiser une variable `som` et écrire une boucle `for`**.**

**b)** Parcourir la liste dans le sens des indices croissants et effectuer le test avec les deux conditions.

**c)** Parcourir la liste dans le sens des indices décroissants.

## 1.6

**a)** Décrire une liste en comparant les points d'indice *i* et $i + 1$ à la moyenne.

**b)** Appeler la fonction `construction_successeurs` qui retourne la liste `successeurs`. Parcourir tous les éléments de la liste `successeurs` sauf le dernier.

**c)** Appeler la fonction `decompose_vagues` qui retourne la liste `vague`. Parcourir tous les éléments de la liste `vague`.

**d)** Parcourir tous les éléments de la liste `proprietes(liste_niveaux)`.

## 1.7

**a)** Utiliser `math.factorial()` et `math.exp()` pour effectuer les calculs.

**c)** Utiliser une boucle `for` pour déterminer le maximum de l'écart. La fonction `abs()` retourne la valeur absolue d'un nombre.

**d)** Utiliser une boucle `while`.

**e)** Appeler la fonction `N` pour obtenir le résultat.

**1.8**

**a)** Un octet correspond à 8 bits.

**c)** Les indices des listes commencent à 0 dans Python. Le premier élément dans la liste des nombres premiers appartenant à l'intervalle $[\![1,n]\!]$ a pour indice 0.

**e)** Importer le module `time` et utiliser `time.time()`. Pour tester si un entier est impair, calculer le reste de la division euclidienne par 2.

# Corrigés des exercices

**1.1**

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

*Première possibilité* : `for i in range(n)`.

*i* varie entre 0 et *n*–1 et permet de récupérer un élément d'indice *i* dans la liste `L`.

```
def moyenne(L):
    # la fonction retourne la moyenne des éléments de L
    # L[i] : float
    som=0                 # initialisation de som à 0
    n=len(L)              # nombre d'éléments dans la liste L
    for i in range(n):    # i varie entre 0 et n-1 inclus
        som=som+L[i]      # on ajoute L[i] à som
    moy=som/n             # on divise par le nombre total de notes
    return moy            # la fonction retourne la valeur de la moyenne
```

Deuxième possibilité : `for elt in L`.

`elt` prend la valeur d'un élément de la liste `L` en commençant par le premier élément d'indice 0.

Cette deuxième solution est pratique à utiliser si on n'utilise pas l'indice de l'élément dans la boucle.

```
def moyenne2(L):
    # la fonction retourne la moyenne des éléments de L
    # L[i] : float
    som=0                 # initialisation de som à 0
    for elt in L:         # on décrit tous les éléments de L
        som+=elt          # on ajoute elt à som
    return(som/len(L))    # on divise par le nombre total de notes
```

Le programme principal s'écrit :

```
L=[13, 8.2, 15, 19]       # exemple de liste de notes
print("Moyennes des notes :",moyenne(L))
```

**b)** La variance (ou écart quadratique moyen) est définie par : $\mathrm{var}(X) = \left\langle \left(X - \langle X \rangle\right)^2 \right\rangle = \frac{1}{N}\sum_{i=1}^{N} \left(X_i - \langle X \rangle\right)^2$.

L'écart-type est défini par $\Delta X = \sqrt{\mathrm{var}(X)}$.

```
def ecart_type(L):
    # la fonction retourne l'écart-type des éléments de L
    # L[i] : float
```

```
    from math import sqrt      # fonction racine carrée du module math
    moy=moyenne(L)             # calcul de la moyenne
    som=0                      # initialisation de som à 0
    n=len(L)                   # nombre d'éléments dans la liste L
    for i in range(len(L)):    # i varie entre 0 inclus et n exclu
        som=som+(L[i]-moy)**2
    return sqrt(som/n)
```

**c)**

```
def moy(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
    som=0               # initialisation de som 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
        som+=L[i]       # on pourrait écrire som=som+L[i]
    som=som/len(L)      # calcul de la moyenne
    return som          # retourne la moyenne de L
```

## 1.2

```
def fact(n):
    # la fonction retourne la factorielle de n(int)
    res=1
    if n>1:                     # pas de boucle for si n=1
        for i in range(1, n):   # i varie entre 1 inclus et n exclu
            res*=(i+1)          # on peut écrire res=res*(i+1)
    return res                  # retourne la factorielle de n
```

On verra au chapitre 7 « Récursivité » une version récursive de cette fonction.

## 1.3

**a)**

```
def calculsuite1(n):
    # la fonction retourne u_n avec n int
    u=1
    if n>=1:
        for i in range(n):    # i varie entre 0 inclus et n exclu
            u=3*u+5
    return u
```

On utilise une variable *u*. À chaque étape de la boucle `for`, on multiplie *u* par 3 et on ajoute 5. On calcule ainsi le terme suivant de la suite. On n'utilise pas de liste pour stocker les termes précédents de la suite, ce qui permet de ne pas encombrer la mémoire de l'ordinateur.

**b)**

```
def calculsuite2(n):
    # la fonction retourne la liste L=[u0, u1, u2…, un]
    L=[1]
    if n>=1:
        for i in range(n):    # i varie entre 0 inclus et n exclu
            L.append(3*L[i]+5)
    return L
```

On verra au chapitre 7 « Récursivité » une version récursive pour les suites définies par récurrence.

## 1.4

**a)**

```
def occupe(L, i):
    return L[i]
```

**b)** On considère une file de longueur 3.

Pour la case d'indice 0, on a deux possibilités.

Pour la case d'indice 1, on a également deux possibilités, soit $2 \times 2 = 4$ files au total.

Pour la case d'indice 2, on a $2 \times 2 \times 2 = 2^3$ files au total.

On a donc $2^n$ files différentes de longueur $n$.

**c)** On considère l'exemple suivant : `L = [True, True, False, True]`.

On a $1101_b = 8 + 4 + 0 + 1 = 13$. On initialise $x$ à 0 et $y$ à 1.

- Première étape de la boucle `for` : $i = 0$. $x = 1$ et $y = 1$.
- Deuxième étape de la boucle `for` : $i = 1$ et $y = 2$. Le programme ajoute 0 à $x$.
- Troisième étape de la boucle `for` : $i = 2$ et $y = 4$. Le programme ajoute 4 à $x$.
- Quatrième étape de la boucle `for` : $i = 3$ et $y = 8$. Le programme ajoute 8 à $x$.

```
def versEntier(L):
    # argument d'entrée : liste L
    # la fonction renvoie l'entier correspondant
    x=0
    y=1
    n=len(L)
    for i in range(n):
        x=x+int(L[n-1-i])*y
        y=y*2
    return x
```

**Remarque**

Une version plus intuitive serait d'écrire :

```
for i in range(n):
    x=x+int(L[n-1-i])*2**i
```

Le programme n'est pas optimisé pour des valeurs de $n$ élevées puisque la complexité est en $O(n^2)$ (voir chapitre 3 « Terminaison, correction, complexité »).

On peut également écrire le programme suivant :

```
def versEntier2(L):
    # argument d'entrée : liste L
    # la fonction renvoie l'entier correspondant
    x,y=0,1
    n=len(L)
    for i in range(n-1, -1, -1):
        x+=int(L[i])*y
        y*=2
    return x
```

$n$ désigne le nombre d'éléments de la liste L. Dans la boucle `for`, $i$ varie entre `len(L)-1` inclus et $-1$ exclu avec un pas égal à $-1$. La variable $i$ prend donc les valeurs suivantes : $n-1$, $n-2$, …., 2, 1, 0.

**d)** Pour un codage sur 3 bits, la valeur maximale est $111_b = 4 + 2 + 1 = 7 = 2^3 - 1$. Cela correspond à `L=[True, True, True]`.

Pour un codage sur $x$ bits, la valeur maximale vaut $2^x - 1$. On doit avoir : $n \leq 2^{taille} - 1$.

On considère l'exemple suivant : *taille* = 4 et `L = [True, True, False, True]`.

On a $1101_b = 8 + 4 + 0 + 1 = 13$.

- Première étape de la boucle `while` : $i = 3$. Le reste vaut 13%2 = 1. Le quotient vaut 13//2 = 6.
- Deuxième étape de la boucle `while` : $i = 2$. Le reste vaut 6%2 = 0. Le quotient vaut 6//2 = 3.
- Troisième étape de la boucle `while` : $i = 1$. Le reste vaut 3%2 = 1. Le quotient vaut 3//2 = 1.
- Quatrième étape de la boucle `while` : $i = 0$. Le reste vaut 3%2 = 1. Le quotient vaut 3//2 = 0.

La condition booléenne à écrire est : `while i>=0`.

On peut écrire également la condition : `while n>0`.

## 1.5

**a)**

```
def moyenne(liste_niveaux):
    # la fonction retourne la moyenne des éléments de liste_niveaux(list)
    som=0
    for elt in liste_niveaux:          # on décrit tous les éléments de
                                       # la liste
        som=som+elt
    return(som/len(liste_niveaux))     # on divise par le nombre d'éléments de
                                       # la liste
```

**b)** Il faut appeler la fonction moyenne et ensuite parcourir les éléments de la liste et effectuer un test avec deux conditions. Si les deux conditions sont vérifiées alors on peut quitter la fonction immédiatement avec l'instruction `return i`.

```
def ind_premier_pzd(liste_niveaux):
    # argument d'entrée : liste_niveaux(list)
    # la fonction retourne un entier
    moy=moyenne(liste_niveaux)
    for i in range(len(liste_niveaux)-1):
        if liste_niveaux[i]>=moy and liste_niveaux[i+1]<=moy:
            return i    # les deux conditions précédentes sont vérifiées
    return -1
```

**Remarques**

L'énoncé précise que l'élément doit être supérieur à la moyenne. C'est sous-entendu supérieur ou égal à la moyenne.

La complexité de la fonction `ind_premier_pzd` est linéaire en $O(n)$, en appelant $n$ le nombre d'éléments de `liste_niveaux`. Si on remplace `moy` par `moyenne(liste_niveaux)` dans la boucle `for`, on calcule la moyenne à chaque étape de la boucle `for` et la complexité est quadratique en $O(n^2)$ (voir chapitre 3).

**c)**

```
def ind_dernier_pzd(liste_niveaux):
    # argument d'entrée : liste_niveaux(list)
    # la fonction retourne un entier
    moy=moyenne(liste_niveaux)
    indice=-2
    for i in range(len(liste_niveaux)-1):
        if liste_niveaux[i]>=moy and liste_niveaux[i+1]<=moy:
            indice=i
    return indice
```

Amélioration : On utilise `liste_niveaux` et la moyenne de la liste comme arguments d'entrée. On parcourt la liste à partir de la fin. L'instruction `return` permet de quitter la fonction dès que les deux conditions sont vérifiées. On a alors une complexité en $O(1)$ dans le meilleur des cas.

Cela suppose que le calcul de la moyenne a déjà été effectué une fois auparavant et stocké dans une variable globale `moy`. On peut améliorer la fonction `ind_premier_pzd` en ajoutant `moy` comme argument d'entrée.

```
def ind_dernier_pzd2(liste_niveaux, moy):
    # argument d'entrée : liste_niveaux(list) et moy(float)
    # moy est la moyenne de liste_niveaux
    # la fonction retourne un entier
    for i in range(len(liste_niveaux)-2, -1, -1):
        if liste_niveaux[i]>=moy and liste_niveaux[i+1]<=moy:
            return i    # on est toujours dans le meilleur des cas
    return -2
```

On note $n$ le nombre d'éléments de la liste. Dans cette boucle, on part de l'avant-dernier élément. Il ne faut pas partir du dernier élément car on aurait un message d'erreur en appelant `liste_niveaux[i+1]`. L'indice $i$ varie entre $n$–2 inclus et –1 exclu avec un pas de –1, c'est-à-dire que $i$ prend les valeurs suivantes : $n$–2, $n$–3,…, 2, 1, 0.

## 1.6

**a)**

```
        if liste_niveaux[i]>m and liste_niveaux[i+1]<m:
            successeurs.append(i+1)     # il faut ajouter l'indice i+1 et non
                                        # l'indice i
```

**b)** On considère `liste_niveaux` = [1,–1,–2,2,–2,–1,6,4,–2,–5]. La fonction `decompose_vagues` doit retourner [[–1, –2, 2],[–2, –1, 6, 4]]. On appelle la fonction `construction_successeurs(liste_niveaux)`. On obtient `L` = [1, 4, 8]. On écrit une boucle sur tous les éléments de la liste `L` sauf le dernier :

- Premier élément de `L` : 1 qui correspond à l'indice pour `liste_niveaux`. On omet les données précédant le premier PND. On ajoute les éléments entre `liste_niveaux[1]` inclus et `liste_niveaux[4]` exclu.
- Deuxième élément de `L` : 8 qui correspond à l'indice pour `liste_niveaux`. On ajoute les éléments entre `liste_niveaux[4]` inclus et `liste_niveaux[8]` exclu.

```
def decompose_vagues(liste_niveaux):
    # argument d'entrée : liste_niveaux (list)
    # la fonction retourne la liste vague
    vague=[]    # liste vide
    L=construction_successeurs(liste_niveaux)
    for i in range(len(L)-1):   # on ne considère pas le dernier élément de L
        vague_prov=[]           # on crée une sous-liste vide
        for j in range(L[i],L[i+1]):    # j varie entre L[i] inclus et L[i+1] exclu
            vague_prov.append(liste_niveaux[j])   # ajout des éléments dans
                                                  # la sous-liste
        vague.append(vague_prov)
    return vague
```

**Remarque**

On peut remplacer la deuxième boucle `for` par une extraction d'éléments dans `liste_niveaux` :

```
def decompose_vagues2(liste_niveaux):
    # argument d'entrée : liste_niveaux (list)
    # la fonction retourne la liste vague
    vague=[]
    L=construction_successeurs(liste_niveaux)
    for i in range(len(L)-1):    # i varie entre 0 inclus et len(L)-1 exclu
        vague.append(liste_niveaux[L[i]:L[i+1]])
    return vague
```

**c)** La fréquence d'échantillonnage vaut 2 Hz. On en déduit la période d'échantillonnage : $T_e = \frac{1}{f_e} = 0{,}5$ s.

On considère `liste_niveaux` = [1, –1, –2, 2, –2, –1, 6, 4, –2, –5]. On utilise la fonction `decompose_vagues` qui retourne `vague` = [ [–1, –2, 2],[–2, –1, 6, 4]].

`proprietes(liste_niveaux)` retourne [[4, 1.5], [8, 2.0]].

```
def proprietes(liste_niveaux):
    # argument d'entrée : liste_niveaux (list)
    # la fonction retourne la liste L
    L=[]
    Te=0.5    # intervalle de temps entre deux points
    vague=decompose_vagues(liste_niveaux)
    for elt in vague:
        Hi=max(elt)-min(elt)
        Ti=len(elt)*Te
        L.append([Hi, Ti])
    return L
```

**d)**

```
def hmax(liste_niveaux):
    # argument d'entrée : liste_niveaux (list)
    # la fonction retourne la hauteur maximale
    prop=proprietes(liste_niveaux)
    val_max=0
    for i in range(len(prop)):
        if prop[i][0]>val_max:
            val_max=prop[i][0]
    return val_max
```

## 1.7

**a)**

```
import math            # module math

def Px(k, n, p):
    # arguments d'entrée : k(int), n(int), p(float)
    lambda2=n*p        # ne pas utiliser lambda, qui est réservé dans Python
    y=lambda2**k/math.factorial(k)*math.exp(-lambda2)
    return y

L1=[]
for k in range(31):    # k varie entre 0 inclus et 31 exclu
   L1.append(Px(k,30,0.1))
print(L1)
```

**b)**

```
def Py(k, n, p):
    # arguments d'entrée : k(int), n(int), p(float)
    y=math.factorial(n)/(math.factorial(k)*math.factorial(n-k))\
    *p**k*(1-p)**(n-k)
    return y
L2=[]
for k in range(31):    # k varie entre 0 inclus et 31 exclu
   L2.append(Py(k,30,0.1))
print(L2)
```

**c)**

```
def Ecart(n, p):
    # arguments d'entrée : n(int), p(float)
    val_max=0
    for k in range(n+1): # k varie entre 0 inclus et n+1 exclu
        ecart=abs(Py(k, n, p)-Px(k, n, p))
        if ecart>val_max:
            val_max=ecart
    return val_max
```

**d)**

```
def N(e, p):
    # arguments d'entrée : e(float), p(float)
    n=1
    while Ecart(n, p)>e: # on n'utilise pas de boucle for car on ne connaît pas
                         # à l'avance le nombre d'étapes
        n+=1             # on incrémente n de 1
    return n
```

**e)**

```
print(N(0.008, 0.075))   # on obtient 1
print(N(0.005, 0.075))   # on obtient 124
```

## 1.8

**a)** Une mémoire vive de 4 Go correspond à $4\times10^9\times8=3,2\times10^{10}$ bits. D'après l'énoncé, un booléen occupe 32 bits. On peut donc stocker $\frac{4\times10^9\times8}{32}=10^9$ booléens dans la mémoire vive.
La valeur maximale de $N$ est $10^9$.

**b)** Il n'y a que deux valeurs pour un booléen. On peut donc coder un booléen sur un seul bit au lieu de 32 bits. Par exemple : 0 = `False` et 1 = `True`. Dans ce cas, on gagne un facteur 32.

**c)** On veut obtenir la liste des nombres premiers dans l'intervalle $[\![1,n]\!]$. Les nombres premiers dans l'intervalle $[\![1,10]\!]$ sont : 2, 3, 5, 7. La fonction `erato_iter(N)` doit retourner : `liste_bool=[False, True,True,False,True,False,True,False,False,False]`.

L'indice $i$ dans le programme ci-dessous désigne l'entier dans l'intervalle $[\![1,n]\!]$. Comme les indices des listes commencent à 0 dans Python, l'entier $i$ correspond à `liste_bool[i - 1]`.

```
import math    # module math

def erato_iter(N):
    # argument d'entrée : N entier >= 1
    liste_bool = [True for j in range(N)]      # j varie entre 0 inclus et N exclu
    liste_bool[0] = False                      # 1 n'est pas un nombre premier
    for i in range(2,int(math.sqrt(N))+1):     # i varie entre 2 inclus et
                                               # int(sqrt(N))+1 exclu
        if liste_bool[i-1] != False:           # l'entier i a pour indice i-1 dans
                                               # liste_bool
            k=2
```

```
            while i*k <= N:
                liste_bool[i*k-1] = False
            # l'entier i*k a pour indice i*k-1 dans liste_bool
            # l'entier i*k n'est pas premier, on le marque comme Faux
                k=k+1
    return liste_bool
```

**d)** Si $x_i$ est impair à chaque opération, on a :

$$A = \sum_{i=1}^{N-1} 2^i = \frac{2^1 - 2^N}{1-2} = 2^N - 2$$

**e)**

```
import time   # module time
def bbs(N):
    # argument d'entrée : N entier >= 1
    p1=24375763
    p2=28972763
    M=p1 * p2
    # calculer la graine
    t=time.time()
    xi=int((t-int(t))*1e7)
    A=0
    for i in range(N):    # i varie entre 0 inclus et N exclu
        if xi%2==1:       # xi est impair
            A = A + 2**i
        # calculer le nouvel xi
        xi = (xi**2) % M
    return A
```

# Représentation graphique

CHAPITRE 2

Plan



### Thèmes abordés dans les exercices

- Bibliothèque `matplotlib`
- Utilisation du module `matplotlib.pyplot`
- Utilisation de la fonction `plot()`

### Points essentiels du cours pour la résolution des exercices

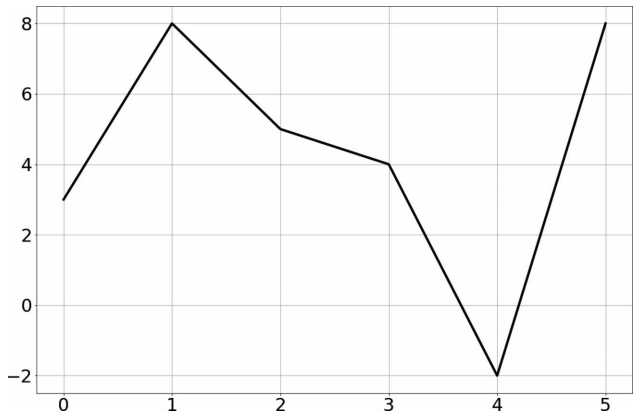
- Représenter graphiquement *y* en fonction de *x*
- Tracer un histogramme

## Les méthodes à retenir

### Graphiques

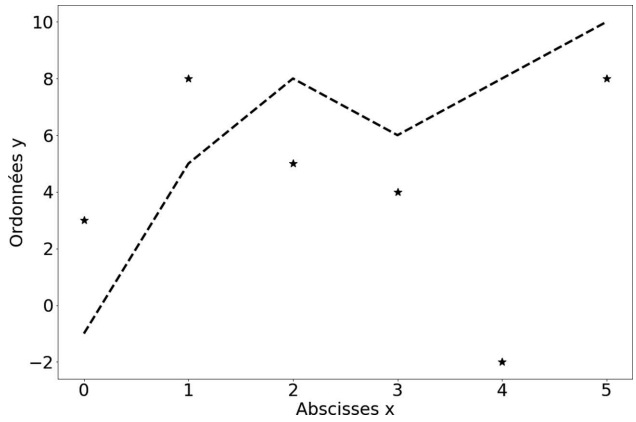
Le module `pyplot` de la bibliothèque `matplotlib` permet d'afficher des graphiques. La fonction `plt.plot(x, y)` permet de représenter graphiquement *y* en fonction de *x*. Les arguments *x* et *y* désignent des listes.

```
import matplotlib.pyplot as plt  # module matplotlib.pyplot renommé plt
x=[0, 1, 2, 3, 4, 5]   # liste de 6 valeurs pour les abscisses
y=[3, 8, 5, 4, -2, 8]  # liste du même nombre de valeurs pour les ordonnées
plt.figure()           # nouvelle fenêtre graphique
plt.plot(x,y)          # représentation graphique de y en fonction de x
plt.grid()             # affiche la grille
plt.show()             # affiche le graphique
```




Les points sont reliés entre eux par des segments de droite. L'instruction `plt.grid()` permet d'afficher la grille. Si on ne veut pas relier les points entre eux, on utilise « * » ou « + ». On peut superposer plusieurs graphes dans une même fenêtre graphique.

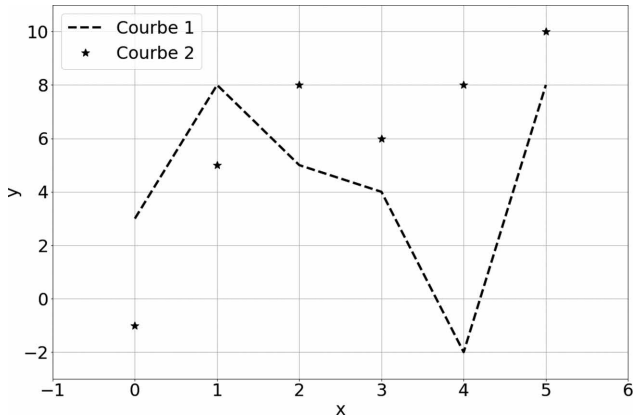
```
y2=[-1, 5, 8, 6, 8, 10]
plt.figure()                  # nouvelle fenêtre graphique
plt.xlabel('Abscisses x')     # axe des abscisses
plt.ylabel('Ordonnées y')     # axe des ordonnées
plt.plot(x, y, '*')           # points non reliés représentés par '*'. On peut mettre '+'
plt.plot(x, y2, '--')         # deux tirets : ligne discontinue
plt.show()                    # affiche le graphique
```




Pour agrémenter les graphiques :

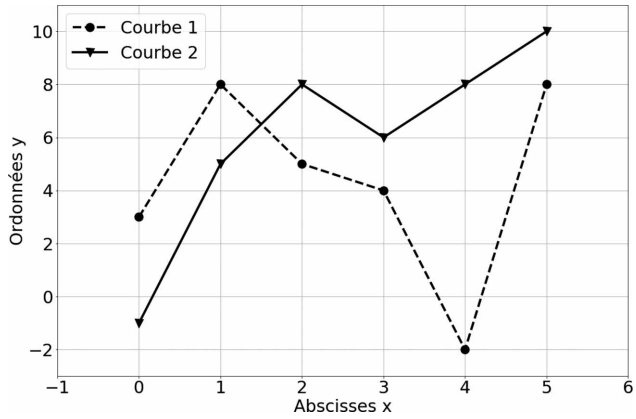
```
plt.figure()          # nouvelle fenêtre graphique
plt.xlabel('x')       # axe des abscisses
plt.ylabel('y')       # axe des ordonnées
plt.title('Titre')    # titre du graphique
plt.plot(x, y, '--')  # ligne discontinue entre les points
plt.plot(x, y2, '*')  # points non reliés représentés par '*'
plt.legend(['Courbe 1', 'Courbe 2'])     # légende des deux courbes
```

```
plt.axis([-1, 6, -3, 11])   # précise les bornes pour les abscisses et les ordonnées
plt.grid()                  # affiche la grille
plt.show()                  # affiche le graphique
```




Dans l'aide en ligne (taper dans la console : `?plt.plot`), on obtient toutes les options pour la couleur, les symboles et l'épaisseur du trait.

```
plt.figure()                          # nouvelle fenêtre graphique
plt.xlabel('Abscisses x')
plt.ylabel('Ordonnées y')
plt.title('Titre du graphique')
plt.plot(x, y, color='r', marker='o', linestyle="--")
plt.plot(x, y2, color='g', marker="v", linestyle="-", linewidth=4)
# color : choix de la couleur ('r' : red, 'g' : green, 'b' : blue, 'black' : black)
# marker : différents symboles ('+', '.', 'o', 'v')
# linestyle : style de la ligne ('-' : ligne continue,
# '--' : ligne discontinue,
# ':' : ligne pointillée)
# linewidth : épaisseur du trait
plt.legend(['Courbe 1', 'Courbe 2'])  # permet de légender les courbes
plt.axis([-1, 6, -3, 11])             # précise les bornes pour les abscisses et les ordonnées
plt.grid()                            # affiche la grille
plt.show()                            # affiche le graphique
```

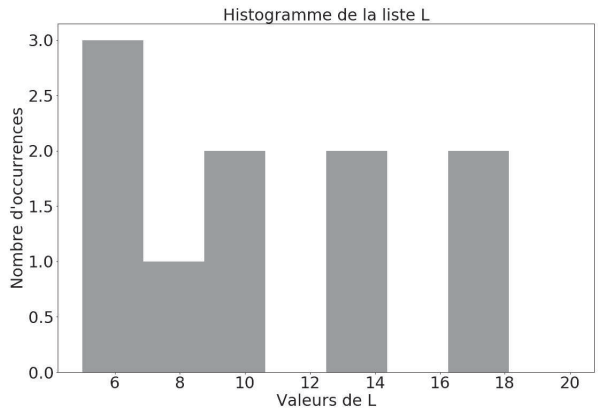


Pour les concours, il faut connaître les fonctions suivantes du module `matplotlib.pyplot` :

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

La fonction `plt.hist()` permet de représenter un histogramme.

```
import matplotlib.pyplot as plt   # module matplotlib.pyplot renommé plt
plt.figure()                      # nouvelle fenêtre graphique
L=[2, 2, 6, 6, 3, 14, 14, 6, 8, 9, 9,17.5,18]
plt.figure()                      # nouvelle fenêtre graphique
plt.hist(L, range=(5, 20), bins=8, color='gray')
plt.title('Histogramme de la liste L')
plt.xlabel('Valeurs de L')
plt.ylabel("Nombre d'occurrences")
    # range=(5, 20) permet de préciser le minimum et le maximum des valeurs
    # représentées sur l'histogramme. Par défaut : range=(min(L), max(L))
    # bins=8=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='gray' permet de préciser la couleur des barres
plt.show()                        # affiche le graphique
```



# Énoncés des exercices

## 2.1

■□□□

**Tracé d'une fonction définie par morceaux**

On considère la fonction *f* définie sur [0, 2[ par :

$$f(x)=\begin{cases} x & \text{pour } 0 \le x < 1 \\ 1 & \text{pour } 1 \le x < 2 \end{cases}$$

**a)** Proposer une fonction `F` qui admet comme argument un réel *x* et retourne $f(x)$.

**b)** Tracer la courbe représentative de la fonction *f* pour *x* variant de 0 à 1,99 inclus avec un pas de 0,01 en utilisant des listes.

Le graphique doit avoir les caractéristiques suivantes :

- Affichage de « x » pour l'axe des abscisses et « y » pour l'axe des ordonnées.
- Affichage du titre : « Graphe y=f(x) ».
- Axe des *y* compris entre 0 et 1,5.

## 2.2

■■■□

**Tracé d'une fonction à deux arguments**

Soit *N* un nombre entier. Par défaut, on pose *N* = 101. On considère alors un échantillonnage de l'intervalle [0, 1] en *N* points. Pour tout entier *n*, on définit sur [0, 1] la fonction $f_n$ par : $f_n(x)=n\sin(n\pi x)^2(1-x)^n$.

**a)** Proposer une fonction `MaxiListe`, d'argument *L*, renvoyant le maximum de la liste *L*, liste contenant des nombres, et le premier indice où ce maximum est atteint. Pour cela, on n'utilisera pas les fonctions et méthodes déjà programmées dans Python et ses modules.

**b)** Proposer une fonction `Sup`, admettant deux arguments :

- une fonction *f* définie sur l'intervalle [0, 1] ;
- le nombre de points d'échantillonnage de l'intervalle [0, 1], noté *N*, sur lesquels seront effectués les calculs.

Cette fonction retourne comme valeurs le maximum de *f* sur les points d'échantillonnage et la plus petite abscisse où ce maximum est réalisé.

**c)** Proposer une fonction `F`, à deux arguments *n* et *x*, renvoyant la valeur $f_n(x)$.

**d)** Proposer une fonction `Graph` qui affiche sur une même figure le graphe sur [0, 1] de toutes les fonctions $f_n$ où *n* appartient à une liste *L* d'entiers naturels passée en argument et *N* est passé en argument.

## 2.3

■■■■

**Tracé d'une fonction définie par rapport à une suite**

On pose *M* = 20 et *m* = 10. À un nombre *c* quelconque, on associe la suite $(u_n)_{n\ge 0}$ définie par :

$$u_0 = 0 \text{ et } u_{n+1} = u_n^2 + c \text{ pour } n \ge 0$$

S'il existe, on note *k* le plus petit entier tel que l'on ait 0 ≤ *k* ≤ m et $|u_k| > M$.

On définit alors la fonction $f$ par :

$$f : c \mapsto \begin{cases} k & \text{s'il existe} \\ m+1 & \text{sinon} \end{cases}$$

**a)** Proposer une fonction `F` qui admet comme argument un nombre $c$ et retourne $f(c)$.

**b)** Tracer l'allure de la courbe représentative de la fonction $f$ sur [–2, 2], en créant une liste de 400 valeurs équiréparties entre –2 et 2 inclus.

### 2.4

■□□□

**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 # module matplotlib.pyplot renommé 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 le graphique
```

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.

## Du mal à démarrer ?

### 2.1

**a)** Définir une fonction qui retourne $f(x)$.

**b)** Utiliser `L.append()` pour ajouter des éléments à la liste `L`.

### 2.2

**a)** Définir deux variables `nbmax` et `imax` et parcourir la liste pour comparer chaque valeur à `nbmax`.

**b)** Définir l'intervalle `dx` entre deux abscisses consécutives.

**c)** Utiliser `def` pour définir une fonction.

**d)** Définir deux boucles pour parcourir les valeurs de `L` et pour définir la liste des abscisses et des ordonnées pour chaque graphe.

**2.3**

**a)** Utiliser une boucle `while`.

**b)** Utiliser `append` pour ajouter des éléments dans les listes `X` et `Y`.

**2.4**

Utiliser `plt.hist` avec `bins=12`.

# Corrigés des exercices

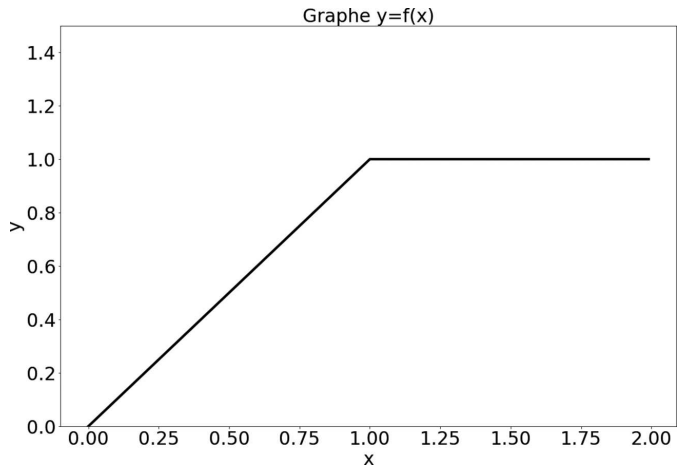
**2.1**

**a)**

```
def F(x):
    # la fonction renvoie F(x) avec x float
    if x>=0 and x<1:
        return x
    elif x>=1 and x<2:
        return 1
    else:
        return 0
```

**b)**

```
import matplotlib.pyplot as plt    # module matplotlib.pyplot renommé plt
n=200                              # nombre de points
x=[i*0.01 for i in range(n)]       # liste des abscisses
y=[]                               # liste vide
for i in range(n):
    y.append(F(x[i]))              # on ajoute F(x[i])

plt.figure()                       # nouvelle fenêtre graphique
plt.xlabel('x')                    # axe des abscisses
plt.ylabel('y')                    # axe des ordonnées
plt.title('Graphe y=f(x)')         # titre du graphique
plt.plot(x, y)                     # graphe y en fonction de x
plt.ylim(0, 1.5)                   # axe des y compris entre 0 et 1.5
plt.show()                         # affiche le graphique
```




## 2.2

**a)**

```
def MaxiListe(L):
    # la fonction renvoie le maximum et l'indice pour la liste L
    nbmax=L[0]
    imax=0
    for i in range(1, len(L)):
        if L[i]>nbmax:
            imax=i
            nbmax=L[i]
    return nbmax, imax
```

**b)**

```
def Sup(f, N):
    # la fonction renvoie ymax et xmax pour la fonction f
    # N est un entier
    a, b=0,1                  # intervalle [a, b]
    dx=(b-a)/(N-1)            # intervalle dx entre deux abscisses
    ymax=f(0)
    xmax=0
    for k in range(1, N):     # k varie entre 1 et N exclu
        x=a+k*dx
        if f(x)>ymax:
            xmax=x
            ymax=f(x)
    return ymax, xmax
```

**c)**

```
def F(n, x):
    # la fonction renvoie fn(x) avec x float et n int
    import math as m   # module math renommé m
    return n*((m.sin(n*m.pi*x))**2)*((1-x)**n)
```

**d)**

```
def Graph (L, N):
    # arguments d'entrée : liste L et N int
    import matplotlib.pyplot as plt # module matplotlib.pyplot renommé plt
    plt.figure()          # nouvelle fenêtre graphique
    for n in L:           # on prend successivement les valeurs de L
        Lx, Ly=[], []     # liste vide des abscisses et des ordonnées
        a, b=0, 1
        dx=(b-a)/(N-1)    # intervalle dx entre deux abscisses
        for i in range(N):
            abscisse=a+i*dx
            Lx.append([abscisse])
            Ly.append([F(n,abscisse)])
        plt.plot(Lx, Ly)  # graphe Ly en fonction de Lx
    plt.show()            # affiche le graphique
```

## 2.3

**a)**

```
def F(c):
    # argument d'entrée : c(float)
    # déclaration de 4 variables locales
    M=20
    m=10
    u=0
    k=0
    while k<=m and abs(u)<=M:
        u=u**2+c
        k+=1
    if k<=m:
        return k
    else:
        return m+1
```

**b)**

```
import matplotlib.pyplot as plt  # module matplotlib.pyplot renommé plt
N=400                            # nombre de points sur le graphique
mini=-2
maxi=2
h=(maxi-mini)/(N-1)
X=[]
Y=[]
for i in range(N):
    X.append(mini+i*h)
    Y.append(F(X[i]))
plt.figure()                     # nouvelle fenêtre graphique
plt.plot(X, Y)                   # graphe Y en fonction de X
plt.show()                       # affiche le graphique
```

## 2.4

```
import matplotlib.pyplot as plt # module matplotlib.pyplot renommé 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 le graphique
```

# Terminaison, correction, complexité

CHAPITRE 3

Plan



### Thèmes abordés dans les exercices

- Terminaison
- Correction
- Complexité

### Points essentiels du cours pour la résolution des exercices

- Démontrer la terminaison d'une fonction
- Savoir définir un variant de boucle
- Démontrer la correction d'une fonction
- Savoir définir un invariant de boucle
- Calculer le nombre d'opérations élémentaires dans le meilleur ou dans le pire des cas
- Modification de fonctions pour diminuer le temps d'exécution
- Évaluer la complexité d'une fonction

## Les méthodes à retenir

### 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):           # arguments d'entrée : liste L et un élément x
    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):
    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 vrai, alors $res = i!$. Dans la boucle, le programme fait le calcul : $res \times (i+1) = i! \times (i+1) = (i+1)!$. *res* prend alors 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.

# 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):
    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 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_max`.

On rencontre les types suivants de complexité :

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

# Énoncés des exercices

## 3.1

**Comparaison de deux listes (Mines Ponts 2017)**

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

**b)** Que peut-on dire de la complexité de cette fonction ?

**c)** Préciser le type de retour de cette fonction.

## 3.2

**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 $\sigma^2$ 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 :

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

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

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

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

**b)** 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
```

**c)** Doit-on s'attendre à une différence de type de la complexité entre une fonction évaluant $S$ et une fonction évaluant $K$ ?

# Du mal à démarrer ?

## 3.1

**a)** Tester si les deux listes ont la même longueur et utiliser une boucle `for` pour parcourir tous les éléments de `L1`.

**b)** On se place dans le pire des cas avec deux listes de même longueur et identiques. Calculer le nombre total d'opérations élémentaires.

**c)** La fonction retourne `True` ou `False`.

## 3.2

**a)** Initialiser une variable `som` et écrire une boucle `for`.

**b)** On a deux boucles `for` imbriquées.

**c)** On a une seule boucle dans les deux cas.

# Corrigés des exercices

## 3.1

**a)**

```
def egal(L1, L2):
    # arguments d'entrée : deux listes L1 et L2
    # la fonction retourne True si les deux listes sont égales, False sinon
    if len(L1)!=len(L2):          # listes de longueurs différentes
        return False
    else:
        for i in range(len(L1)):  # on 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
```

**b)** On se place dans le pire des cas. On suppose que `len(L1)` = `len(L2)` = $n$.

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 $3 + 3n$ 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.

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

## 3.2

**a)**

```
def moyenne(L):
    # la fonction retourne la 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)
```

**b)** 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 skewness(liste_hauteurs):
    # argument d'entrée : liste_hauteurs (list)
    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
```

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

# Algorithmes, recherche dans une liste, recherche par dichotomie

CHAPITRE 4

Plan



### Thèmes abordés dans les exercices

- Recherche d'un élément dans une liste non triée
- Recherche d'un mot dans une chaîne de caractères
- Recherche par dichotomie dans une liste triée
- Recherche par dichotomie du zéro d'une fonction continue et strictement monotone

### Points essentiels du cours pour la résolution des exercices

- Savoir parcourir les éléments d'une liste pour réaliser les tests
- Savoir partager une liste en deux en réduisant par deux l'intervalle
- Faire la différence pour la méthode de dichotomie pour une liste et le zéro d'une fonction
- Utiliser des boucles `for` ou `while`

## Les méthodes à retenir

### Recherche d'un élément dans une liste

La méthode est de parcourir les éléments de la liste jusqu'à ce qu'on trouve l'élément recherché.

La fonction ci-dessous convient pour les listes qui ne sont pas nécessairement triées.

```
def recherche(L, x):
    for elt in L:      # on parcourt les éléments de la liste L
        if x==elt:     # on teste si x=elt
            return True       # return provoque un arrêt de la boucle
                              # et une sortie de la fonction
    return False
```

On peut utiliser une boucle `while` :

```
def recherche(L, x):
    i=0
    while i<len(L):
        if L[i]==x:
            return True       # return provoque un arrêt de la boucle
                              # et une sortie de la fonction
        else:
            i=i+1
    return False
```

Le programme suivant affiche à l'écran si un élément est présent dans une liste :

```
L=[3,-5,8,11,4]          # L est de type list
print(recherche(L,4))    # retourne False
```

# Recherche par dichotomie dans une liste triée

La méthode de dichotomie 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…

Le programme suivant affiche à l'écran si un élément est présent dans une liste triée par ordre croissant.

```
def dicho(L, x):    # L est une liste, x = élément recherché
    debut=0         # indice de début
    fin=len(L)-1    # indice de fin
    rep=False
    while fin>=debut and rep==False:
        milieu=(debut+fin)//2    # milieu doit être un entier
        if x==L[milieu]:
            rep=True             # l'élément recherché est trouvé
        elif x>L[milieu]:        # x est dans la deuxième moitié
            debut=milieu+1       # on réduit l'intervalle
        else:                    # x est dans la première moitié
            fin=milieu-1         # on réduit l'intervalle
    return rep

L=[3,5,8,12,15,16,18]
print(dicho(L,8))                # affiche True
print(dicho(L,3))                # affiche True
print(dicho(L,55))               # affiche False
```

On considère deux indices `debut` et `fin` qui déterminent l'intervalle de recherche d'un élément dans la liste.

Exemple de fonctionnement de l'algorithme avec `x` = 8 :

- 1re étape de la boucle `while` : `debut` = 0, `fin` = 6 et `milieu` = 3 = `int(6/2)`. On compare *x* avec `L[3]` = 12. Comme *x* < `L[3]`, il suffit de chercher dans la première moitié de l'intervalle [0, 6] dans l'étape suivante. À la fin de la première étape, `fin` prend la valeur 3 – 1 = 2.
- 2e étape de la boucle `while` : `debut` = 0 et `fin` = 2. Le milieu vaut `int(2/2)` = 1. On compare *x* avec `L[1]` = 5. Comme *x* > `L[1]`, il suffit de chercher dans la deuxième moitié de l'intervalle [0, 2] dans l'étape suivante. À la fin de la deuxième étape, `debut` prend la valeur 1 + 1 = 2.
- 3e étape de la boucle `while` : `debut` = 2 et `fin` =2. On a trouvé l'élément *x*.

# Énoncés des exercices

## 4.1

■□□□

**Recherche du minimum dans une liste non triée**

On considère `L` une liste non vide et non triée contenant des éléments de type `float`.

**a)** Écrire une fonction `rec_min` qui admet comme argument une liste `L`. Cette fonction retourne le minimum des éléments.

**b)** Écrire une fonction `rec_min_ind` qui admet comme argument une liste `L`. Cette fonction retourne l'indice et la valeur du minimum de L.

**c)** Écrire le programme principal permettant d'afficher l'indice et la valeur du minimum pour la liste `L2=[3, 8.2, 5, 19, -2]`.

## 4.2

■□□□

**Recherche d'un élément dans une liste non triée, algorithme naïf, complexité**

**a)** Écrire une fonction `rec_pos` qui admet comme arguments une liste et un élément à rechercher. Cette fonction retourne `True` si l'élément est présent dans la liste ainsi que l'indice de la première occurrence dans la liste. Si l'élément n'est pas présent, cette fonction retourne `False` et `-1`.

**b)** Évaluer la complexité de cet algorithme pour une liste de longueur *n* dans le cas le moins favorable.

## 4.3

■■■□

**Recherche d'un mot dans une chaîne de caractères**

Écrire une fonction `rec_mot` qui admet comme arguments une chaîne de caractères et un mot. Cette fonction retourne `True` si le mot est présent dans la chaîne de caractères ainsi que l'indice de la première lettre du mot dans la chaîne de caractères. Si le mot n'est pas présent, cette fonction retourne `False` et –1.

## 4.4

■■□□

**Recherche dichotomique, complexité**

On considère la liste triée par ordre croissant : `L=[-3.5, -2.8, 0, 15.3, 18, 25, 29]`.

**a)** Écrire une fonction `rec_dicho` qui admet comme arguments une liste triée par ordre croissant et un élément à rechercher. Cette fonction utilise la méthode de dichotomie et retourne `True` si l'élément recherché est présent dans la liste, sinon retourne `False`.

**b)** Évaluer la complexité de cet algorithme pour une liste de longueur n»1 dans le pire des cas. On considère que l'entier *n* est une puissance de 2.

# Du mal à démarrer ?

**4.1**

**a)** Définir une variable `val_min` et parcourir la liste pour comparer chaque valeur à `val_min`.

**b)** Définir deux variables `indice` et `val_min`. Parcourir la liste.

**c)** Appeler une seule fois la fonction `rec_min_ind` et stocker le résultat dans une variable.

**4.2**

**a)** Utiliser `while` pour parcourir les éléments de la liste et tester si l'élément recherché est présent.

**b)** Calculer le nombre d'opérations élémentaires à chaque itération.

**4.3**

Utiliser deux variables pour chercher l'indice du premier caractère du mot présent dans la chaîne de caractères et pour tester si les autres caractères du mot recherché sont présents dans la chaîne de caractères.

**4.4**

**a)** Définir un intervalle de recherche de l'élément dans la liste. Utiliser une boucle `while` et réduire de moitié l'intervalle à chaque étape.

**b)** Calculer le nombre d'opérations élémentaires à chaque itération.

# Corrigés des exercices

**4.1**

**a)** On définit une variable `valmin` qui prend comme valeur le premier élément de `L`. On parcourt la liste à partir du deuxième élément et on compare à chaque étape `L[i]` à `valmin`.

```
def rec_min(L):
    # la fonction retourne le minimum des éléments de la liste L
    valmin=L[0]
    for i in range(1,len(L)):   # i varie entre 1 inclus et len(L) exclu
        # inutile de commencer la boucle for à 0
        if L[i]<valmin:
            valmin=L[i]
    return valmin
```

**Remarque**

La fonction `min(L)` renvoie le minimum de la liste `L`. Dans de nombreux problèmes de concours, l'énoncé demande d'écrire cette fonction sans utiliser la fonction `min`.

**b)**

```
def rec_min_ind(L):
    # la fonction retourne l'indice et le minimum des éléments de la liste L
    val_min=L[0]
```

```
    indice=0
    for i in range(1, len(L)):
        if L[i]<val_min:
            val_min=L[i]
            indice=i
    return indice,val_min
```

**c)** La fonction `rec_min_ind` retourne un tuple `res` contenant deux éléments. On récupère le premier élément avec `res[0]` et le deuxième élément avec `res[1]`.

```
L2 = [3, 8.2, 5, 19, -2] # mettre un point pour les nombres flottants
res=rec_min_ind(L2)      # le résultat de la fonction est stocké dans un tuple
print('Indice du minimum :', res[0])  # première valeur du tuple : res [0]
print('Valeur du minimum :', res[1])  # deuxième valeur du tuple : res [1]
```

## 4.2

**a)**

```
def rec_pos(L, x):
    # arguments d'entrée L(list) et x
    # la fonction retourne True si x est dans L et l'indice de
    # la première occurence dans la liste L
    i=0  # initialisation de l'indice i
    n=len(L)
    while i<n:              # ne pas oublier ":" à la fin de la ligne
        if x==L[i]:         # ne pas oublier ":" à la fin de la ligne
            return True, i # return provoque un arrêt de la boucle
                           # et une sortie de la fonction
        i+=1               # on pourrait écrire i=i+1
    return False, -1       # on pourrait écrire (False, -1)
```

**Remarque**

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

**b)**

On appelle *n* le nombre d'éléments de la liste `L`. On note O(*n*) la complexité de la fonction `rec_pos(L, x)`.

On cherche à calculer le nombre d'opérations élémentaires :

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

La boucle `while` est exécutée *n* fois dans le pire des cas (si l'élément n'est pas présent dans la liste). Le nombre d'opérations élémentaires vaut : 3 + 3*n*. La complexité est linéaire en O(*n*) pour la fonction `rec_pos`.

## 4.3

```
def rec_mot (L, elt):  # arguments d'entrée L et elt
    rep=False
    position=0
```

```
    i=0        # recherche de l'indice du premier caractère de elt dans L
    while i<=(len(L)-len(elt)) and rep==False:
        j=0    # teste si les autres caractères de elt sont présents dans L
        while j<=(len(elt)-1) and elt[j]==L[i+j]:
            j+=1
        if j==len(elt): # elt est bien présent dans L
            position=i
            rep=True
        i+=1
    return rep, position
```

Le programme suivant donne des exemples d'utilisation de cette fonction :

```
C1= "test présence dans une chaîne"    # chaîne de caractères
elt="présence"             # on peut écrire elt='présence'
print(rec_mot(C1, elt))    # affiche (True, 5)
print(rec_mot(C1, 'dans')) # affiche (True, 14)
print(rec_mot(C1, 's'))    # affiche (True, 2)
print(rec_mot(C1, 'test')) # affiche (False, 0)
```

On considère la chaîne de caractères `C1` = "test présence dans une chaîne" et un mot `elt` = "présence".

La longueur de `C1` vaut `len(C1)` = 29. La longueur de `elt` vaut `len(elt)` = 8.

On définit un indice *i* qui correspond à l'indice du premier caractère du mot s'il est présent dans `C1`.

Il est inutile de faire varier *i* entre 0 et 28 mais uniquement entre 0 et `len(L)` – `len(elt)` = 21.

Lorsque *i* atteint 5, on a `C1[i]` = 'p' : c'est bien le premier caractère du mot "présence".

Ensuite, il faut une deuxième boucle en faisant varier *j* entre 0 et `len(elt)` – 1 pour tester si tous les caractères de `elt` sont présents dans `C1`.

## 4.4

**a)**

```
def rec_dicho(L, elt):
    # arguments d'entrée L(list) et x
    # la fonction retourne True si x est dans L, False sinon
    debut=0
    fin=len(L)-1    # intervalle de recherche dans L
    rep=False
    while fin>=debut and rep==False:
        milieu=(debut+fin)//2    # milieu doit être un entier
        if elt==L[milieu]:
            rep=True             # elt est trouvé
        elif elt >L[milieu]:     # elt est dans la deuxième moitié
            debut=milieu+1       # on réduit l'intervalle
        else:                    # elt est dans la première moitié
            fin=milieu-1         # on réduit l'intervalle
    return rep
```

**b)** On se place dans le pire des cas pour évaluer la complexité, c'est-à-dire dans le cas où l'élément n'est pas présent dans la liste de longueur *n*. Pour simplifier les calculs, on considère que l'entier *n* est une puissance de 2.

Trois affectations, un calcul : quatre opérations élémentaires.

- Calcul du nombre d'opérations élémentaires à chaque itération :
- trois comparaisons : trois opérations élémentaires ;
- calcul du milieu et affectation : deux opérations élémentaires ;
- `if elt==L[milieu]` : deux opérations élémentaires (appel de `L[milieu]` et test) ;
- `elt>L[milieu]` : deux opérations élémentaires (appel de `L[milieu]` et test) ;
- une affectation : une opération élémentaire.

Le nombre d'opérations élémentaires vaut 10 à chaque itération.

- Pour chaque itération, on divise par deux la longueur de la liste.

Après la première itération, la longueur de la liste est $\frac{n}{2}$. Après la deuxième itération, la longueur de la liste est $\frac{n}{2^2}$. Après k itérations, on arrive à une liste de longueur 1. La longueur de la liste est $\frac{n}{2^k}=1$. On a alors In $(n)=k$ In $(2)$, soit : $k=\frac{\ln(n)}{\ln(2)}=\log_2(n)$.

- Le nombre d'opérations élémentaires vaut donc $4+10k$ dans le pire des cas. Il est donc proportionnel à $\log_2(n)$.

La complexité est logarithmique en $O(\log_2(n))$. On peut la noter également : $O(\ln(n))$.

**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.

# Lecture et écriture de fichiers

CHAPITRE 5

Plan



### Thèmes abordés dans les exercices

- Écriture de fichiers
- Ajouter un saut de ligne
- Ajouter des éléments dans un fichier
- Lecture de fichiers
- Supprimer un saut de ligne
- Séparer une chaîne de caractères en utilisant le séparateur ";"

### Points essentiels du cours pour la résolution des exercices

- Utilisation de `open()`
- Différence entre `readlines()` et `readline()`
- Utilisation de `split(';')`

## Les méthodes à retenir

### Création d'un fichier texte

L'instruction `f=open('fichier.txt','w')` permet de créer un fichier texte en mode écriture (`'w'` pour *write*). Pour ajouter des caractères à une ligne, on utilise l'instruction `f.write(chaine)` où `chaine` désigne une chaîne de caractères. Pour indiquer un saut de ligne, on écrit `f.write('\n')`. Il ne faut pas oublier `f.close()` pour fermer le fichier.

```
x=[25, 20, 15, 12, 9]
y=[10.9, 9.3, 8.2, 7.5, 6.2]
n=len(x)                    # longueur de la liste x
f=open('fichier1.txt','w') # création du fichier en écriture : write
```

```
f.write('x;y'+'\n')         # première ligne avec saut de ligne \n
                            # séparateur ;
for i in range(n):
    ligne=str(x[i])+';'+str(y[i])+'\n'
                            # séparateur ; et saut de ligne \n
                            # conversion des float en str
    f.write(ligne)
f.close()                   # fermeture du fichier
```

Le programme crée un fichier texte `'fichier1.txt'` contenant 6 lignes. La première ligne contient la dénomination des deux colonnes (`x` et `y`) séparées par `";"`. Il faut ajouter `'\n'` pour avoir un saut de ligne. Cette première ligne n'est pas obligatoire dans les fichiers .txt.

## Lecture d'un fichier texte avec `readlines()`

L'instruction `f=open('fichier.txt', 'r')` permet d'ouvrir un fichier texte en mode lecture (`'r'` pour *read*). L'instruction `data=f.readlines()` permet de récupérer l'ensemble des données dans une liste `data`. Chaque élément de `data` correspond à une ligne du fichier texte. Pour récupérer les données contenues dans chaque ligne, il faut enlever les caractères d'échappement (ici saut de ligne) en utilisant `strip()`. La méthode `split(';')` permet de séparer la ligne en une liste de mots avec le séparateur `";"`. On récupère les données dans deux listes `X` et `Y`.

```
f=open('fichier1.txt', 'r')     # ouverture du fichier en lecture : read
data=f.readlines()              # on récupère toutes les lignes
                                # du fichier dans une liste data
n=len(data)                     # nombre d'éléments de data
X=[]
Y=[]

for i in range(1, n):           # la première ligne ne contient pas de données
    ligne=data[i]               # on obtient une chaîne de caractères
                                # correspondant à une ligne du fichier texte
    ligne2=ligne.strip()        # on enlève le saut de ligne
    a, b=ligne2.split(';')      # on sépare en deux chaînes de caractères
    X.append(float(a))
    Y.append(float(b))
f.close()                       # fermeture du fichier
```

On obtient deux listes `X` et `Y` contenant les valeurs des listes `x` et `y` du programme précédent.

## Lecture d'un fichier texte avec `readline()`

`f.readlines()` permet de récupérer l'ensemble des données du fichier dans une liste. `f.readline()` permet de lire les lignes du fichier les unes après les autres. On ne connaît pas à l'avance le nombre de lignes du fichier. L'instruction `while 1:` permet d'écrire une boucle infinie. Le test `ligne==''` permet de quitter la boucle `while` à la fin du fichier.

```
f=open('fichier1.txt', 'r')    # ouverture du fichier en lecture : read
L1=[]                          # liste vide
L2=[]                          # liste vide
f.readline()                   # lecture de la première ligne
while 1:                       # boucle infinie
    ligne=f.readline()         # on lit les lignes les unes après les autres
    if ligne=="":
        break                  # on quitte la boucle while en fin de fichier
    ligne2=ligne.strip()       # on enlève le saut de ligne
    a, b=ligne2.split(';')     # on sépare en deux chaînes de caractères
    L1.append(float(a))
    L2.append(float(b))
f.close()                      # fermeture du fichier
```

Les fonctions et méthodes suivantes de gestion de fichiers seraient fournies dans un problème de concours :

| | |
|---|---|
| `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*) |
| `f.readline()` | Lecture d'une ligne du fichier `f` |
| `f.readlines()` | Récupération des données du fichier dans une liste |
| `f.read(n)` | Lecture de `n` caractères du fichier `f` |
| `f.read()` | Lecture de tout le fichier |
| `\n` | Caractère d'échappement : saut de ligne |
| `c1.strip()` | Renvoie une chaîne sans les espaces ni les caractères d'échappement (saut ou fin 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 la chaîne de caractères `'exemple'` |
| `f.close()` | Ferme le fichier `f` |

# Énoncés des exercices

## 5.1

**Création d'un fichier texte contenant des valeurs aléatoires**

Proposer une fonction `fictexte` qui admet comme arguments une chaîne de caractères représentant le nom du fichier et le nombre de lignes *n*. La fonction crée un fichier de *n* valeurs aléatoires entre 0 et 1. On pourra utiliser la fonction `random()` du module `random` qui renvoie une valeur aléatoire entre 0 et 1.

Rappels 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)
\n                 # caractère d'échappement : saut de ligne
f.write('exemple') # écrit dans le fichier f la chaîne de caractères 'exemple'
f.close()          # ferme le fichier f
```

## 5.2

**Création d'une liste de flottants à partir d'un fichier (Mines Ponts 2018)**

On s'intéresse à des mesures de niveau de la surface libre de la mer effectuées par une bouée. Cette bouée contient un ensemble de capteurs incluant un accéléromètre vertical qui fournit, après un traitement approprié, des mesures à étudier. Une campagne de mesures a été effectuée. Les caractéristiques de cette campagne sont les suivantes :

- durée de la campagne : 15 jours ;
- durée d'enregistrement : 20 min toutes les demi-heures ;
- fréquence d'échantillonnage : 2 Hz.

Les relevés de la campagne de mesure sont inscrits dans un fichier texte dont le contenu est défini comme suit. Les informations relatives à la campagne sont rassemblées sur la première ligne du fichier, séparées par des points-virgules (« ; »). On y indique différentes informations importantes comme le numéro de la campagne, le nom du site, le type du capteur, la latitude et la longitude de la bouée, la date et l'heure de la séquence. Les lignes suivantes contiennent les mesures du déplacement vertical (m). Chaque ligne comporte 8 caractères dont le caractère fin de ligne. Par exemple, on trouvera dans le fichier texte les 3 lignes suivantes :

- +0.4256
- +0.3174
- –0.0825
- …

Rappels pour la gestion des fichiers :

```
f=open('fichier.txt', 'w') #'fichier.txt' désigne le nom du fichier f
         # 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
f.close()       # ferme le fichier f
```

**a)** On suppose que chaque caractère est codé sur 8 bits. En ne tenant pas compte de la première ligne, déterminer le nombre d'octets correspondant à 20 minutes d'enregistrement à la fréquence d'échantillonnage de 2 Hz.

**b)** En déduire le nombre approximatif (un ordre de grandeur suffira) d'octets contenus dans le fichier correspondant à la campagne de mesure définie précédemment. Une carte mémoire de 1 Go est-elle suffisante ?

**c)** Si, dans un souci de réduction de la taille du fichier, on souhaitait ôter un chiffre significatif dans les mesures, quel gain relatif d'espace mémoire obtiendrait-on ?

**d)** Les données se trouvent dans le répertoire de travail sous la forme d'un fichier `donnees.txt`. Proposer une suite d'instructions permettant de créer à partir de ce fichier une liste de flottants `liste_niveaux` contenant les valeurs du niveau de la mer. On prendra garde à ne pas insérer dans la liste la première ligne du fichier.

# Du mal à démarrer ?

## 5.1

`f.open('fichier.txt', 'w')` permet de créer un fichier en écriture. Utiliser une boucle `for` pour ajouter une ligne à chaque étape avec `f.write()`.

## 5.2

**a)** 1 octet correspond à 8 bits. Calculer la période d'échantillonnage et le nombre d'échantillons.

**d)** Ouvrir le fichier en lecture et parcourir toutes les lignes du fichier. La méthode `strip()` permet d'enlever le caractère de saut de ligne.

# Corrigés des exercices

## 5.1

Il faut bien faire attention à l'indentation. À chaque étape de la boucle `for`, on ajoute une chaîne de caractères obtenue par la concaténation de deux chaînes de caractères : `str(y)` et `'\n'`. La chaîne de caractères `'\n'` permet d'avoir un saut de ligne.

```
def fictexte(nom, n):          # nom = nom du fichier
                               # et n = nombre de lignes
    import random              # import de la bibliothèque random
    f=open(nom, 'w')           # création du fichier en écriture : write
    for i in range(n):         # i varie entre 0 inclus et n exclu
        y=random.random()      # variable aléatoire entre 0 et 1
        ligne=str(y)+'\n'      # concaténation de deux chaînes de caractères
                               # conversion de y en str ; saut de ligne \n
```

```
        f.write(ligne)       # on ajoute ligne qui est une chaîne de caractères
    f.close()                # fermeture du fichier
```

## 5.2

**a)** La durée totale d'acquisition vaut 20 minutes. La fréquence d'acquisition vaut 2 Hz. On a donc $20 \times 60 \times 2$ échantillons. D'après l'énoncé, chaque échantillon correspond à une ligne du fichier texte, soit 8 caractères. Chaque caractère étant codé sur 8 bits (ou 1 octet), on en déduit qu'il faut au minimum $20 \times 60 \times 2 \times 8 \times 8$ bits, soit $20 \times 60 \times 2 \times 8$ octets.

Pour 20 minutes d'enregistrement, il faut donc au minimum 19,2 ko.

**b)** La campagne de mesure dure 15 jours. Pour chaque demi-heure, il faut 19,2 ko. Il faut donc pour la campagne de 15 jours une taille mémoire minimale de $15 \times 24 \times 2 \times 19{,}2 = 14$ Mo. La carte mémoire de 1 Go est largement suffisante.

**c)** Si on enlève un chiffre significatif à chaque ligne, on utilise 7 octets au lieu de 8. On a une réduction de 1/8 = 13 % de la taille du fichier total.

**d)**

```
liste_niveaux=[]               # création d'une liste vide
f=open('donnees.txt', 'r')     # ouverture du fichier en lecture
data=f.readlines()             # data contient toutes les lignes du fichier
for i in range(len(data)):
    if i>0:                    # on exclut la première ligne du fichier
        val=data[i].strip()    # on enlève les caractères d'échappement
        liste_niveaux.append(float(val))   # conversion en flottant
f.close()                                  # fermeture du fichier
```

# Piles, files, deques, dictionnaires

CHAPITRE 6

Plan



### Thèmes abordés dans les exercices

- Principe LIFO, FIFO
- Passage par référence des listes, deques et dictionnaires dans les fonctions
- Manipulation de piles, files, deques et dictionnaires

### Points essentiels du cours pour la résolution des exercices

- Savoir empiler et dépiler un élément dans une pile
- Savoir enfiler et défiler un élément dans une file
- Savoir ajouter un élément à chaque extrémité d'une deque
- Utilisation des dictionnaires

## Les méthodes à retenir

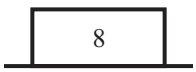
### Principe de fonctionnement d'une pile

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.

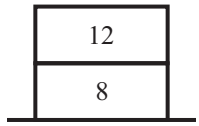
La fonction `empiler` (ou `push`) consiste à ajouter un élément (une assiette dans l'exemple) sur le dessus de la pile.

On empile l'élément 8 à une pile `P` initialement vide. On obtient :




pile P

On empile l'élément 12. On ajoute cet élément sur le dessus de la pile :




pile P

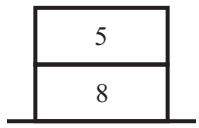
On ne peut pas ajouter directement un élément entre 8 et 12. Pour cela, il faut dépiler, c'est-à-dire enlever l'élément situé en haut de la pile. On utilise la fonction `dépiler` (ou `pop`).

On a bien une structure LIFO puisque le dernier élément entré est le premier sorti.

On obtient après avoir dépilé un élément de la pile :

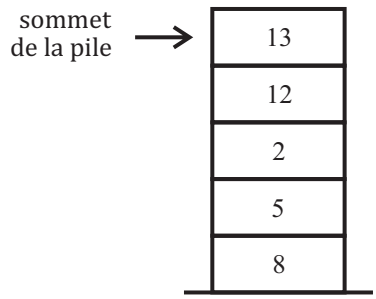

pile P

On empile l'élément 5. On obtient :




pile P

On empile les éléments 2, 12 et 13. On obtient alors :




pile P

Les éléments de la pile sont stockés selon une adresse croissante. On ne connaît que l'adresse du sommet de la pile et on ne peut pas connaître les autres éléments sans dépiler les éléments au-dessus.

**Conséquences pratiques :**

- On ne connaît pas la longueur d'une pile.
- On ne peut pas récupérer un élément quelconque de la pile mais uniquement l'élément situé en haut de la pile.
- Pour récupérer un élément quelconque de la pile, il faut dépiler tous les éléments au-dessus.

# Utilisation de listes

On définit une liste vide `P` qui représente la pile :

```
P=[]            # création d'une liste vide avec Python
```

On empile l'élément `x` dans la pile `P` en utilisant la méthode `P.append()` :

```
P.append(x)     # ajout de l'élément x dans la liste P
```

On teste si la pile `P` est vide :

```
P==[]           # retourne True si la pile est vide sinon retourne False
```

On dépile la pile `P` en utilisant la méthode `P.pop()` :

```
if (P==[]):
    print("On ne peut pas dépiler une pile vide.")
else:
    x=P.pop()  # on enlève l'élément en haut de la pile et on récupère sa valeur dans x
```

# Principe de fonctionnement d'une file

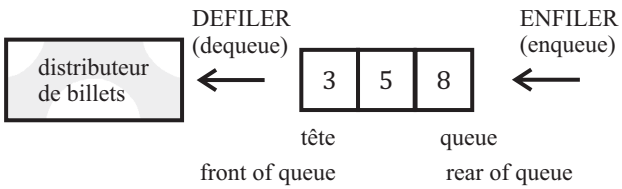
Une file (*queue* en anglais) est une structure de données qui utilise le principe FIFO (*First In, First Out :* « premier entré, premier sorti »).

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).

### Fonction enfiler

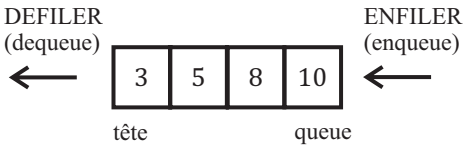
La fonction `enfiler` (*enqueue*) consiste à ajouter un élément à la queue de la file (on dit aussi à l'arrière de la file d'attente).

Soit une file d'attente `F` contenant trois éléments :

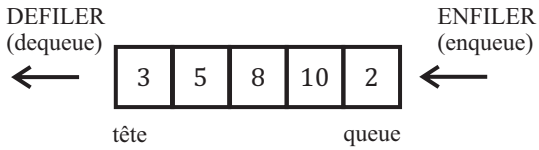



```
def enfiler(F, x):
    F.append(x)    # on ajoute l'élément x à la liste F
```

On enfile l'élément 10 à la file F. On obtient alors :



On enfile l'élément 2, on obtient :

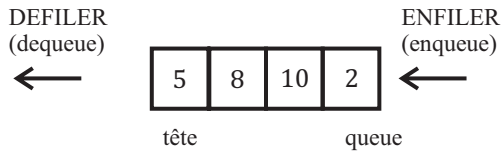



On modélise la file d'attente par une liste Python : `F = [3, 5, 8, 10, 2]`. On dit que 3 est à la tête de la file `F` et 2 est à la queue de la file `F`.

### Fonction défiler

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 entré est le premier sorti.

On obtient alors la file :




```
def défiler (F):
    x=F.pop(0)    # on enlève le premier élément de la liste F
    return x      # retourne la valeur du premier élément de la liste F
```

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 6.4 « Utilisation des deques », le module `collections.deque` qui est conçu pour ajouter et supprimer rapidement des éléments aux deux extrémités.

# Principe de fonctionnement d'une deque

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
```

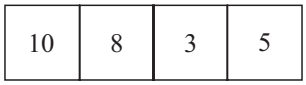
Création d'une deque vide :

```
D=deque()
```

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 :




**Attention**

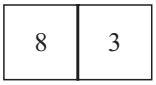
On a un nouveau type : `deque`. Ne pas confondre avec le type `list`.

```
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`.

Création d'une deque :

```
D1=deque([3, 8, 5])    # création de la deque D1
```

Parcours de tous les éléments de la deque `D1` :

```
for elt in D1:
    print(elt)
```

Dans Python, les deques (comme les listes et les dictionnaires) sont des objets muables. On considère l'instruction suivante :

```
E=D
```

Si on modifie `E`, alors `D` est également modifié puisque `D` et `E` font référence à la même adresse mémoire.

```
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`. La fonction `deepcopy()` serait rappelée dans un problème de concours.

```
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 modifiée puisque D ne pointe pas
                    # vers la même adresse mémoire que E
print('Teste si D = E :', D==E)    # affiche False
```

# Principe de fonctionnement d'un dictionnaire

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.

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.

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)`.

*Création d'un dictionnaire vide :*

```
dico_classe={}            # dictionnaire vide avec des accolades {}
```

*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`.

En revanche, on peut modifier une valeur :

```
dico_classe["MPSI"]=45
print(dico_classe)        # {'MPSI': 45, 'MP': 46}
```

*Suppression d'un élément :*

```
del dico_classe["MPSI"]   # suppression de la clé "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
```

Les clés d'un dictionnaire ne sont pas obligatoirement des chaînes de caractères. On va voir plusieurs méthodes pour parcourir un dictionnaire.

*Parcours des clés d'un dictionnaire comme pour une liste :*

```
dico={3:5, 8:5}    # les clés du dictionnaire doivent être différentes
for clé in dico:   # on parcourt les clés de dico
    print(clé)
```

On obtient alors :

```
3
8
```

On peut utiliser également `.keys()` :

```
for clé in dico.keys():                 # on parcourt les clés de dico
    print(clé)
```

*Parcours des clés et valeurs d'un dictionnaire avec* `.items()` *:*

```
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
```

On peut utiliser `.items()` avec `clé`, `valeur` pour dépaqueter le tuple :

```
for clé, valeur in dico_classe.items():
    print("clé :", clé, "valeur :", valeur)
```

**Attention**

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`.

On peut récupérer une liste contenant les clés du dictionnaire :

```
L1=dico.keys()
```

*Copie d'un dictionnaire :*

La fonction `copy()` du module `copy` est à connaître.

```
import copy                 # module copy
d2=d
d3=copy.copy(d)             # copie superficielle de d
d['MP']=48
print(d['MP'])              # la valeur vaut 48
print(d2['MP'])             # la valeur vaut 48
print(d3['MP'])             # la valeur vaut 46
```

L'instruction `d2=d` n'a pas réalisé une copie de `d` puisque `d2` et `d` pointent vers la même adresse mémoire.

Si on modifie un élément du dictionnaire `d`, alors cet élément est modifié dans `d2`.

En revanche, la modification n'apparaît pas dans `d3` puisque `d` et `d3` pointent vers des adresses mémoire différentes.

Les dictionnaires sont des objets muables (voir chapitre 1 « Type, listes, boucles, fonctions »).

**Remarque**

On rencontre deux types de copies pour les objets muables (listes, dictionnaires, deques…) :

- La fonction `copy()` réalise une copie superficielle. Les valeurs des clés sont bien copiées s'il n'y pas de structure imbriquée (liste par exemple). Si les valeurs d'un dictionnaire sont des listes, alors l'adresse mémoire des listes est copiée.
- La fonction `deepcopy()` réalise une copie profonde pour les structures imbriquées. Si les valeurs sont des listes, alors la copie profonde copie bien les listes imbriquées.

# Passage par référence des listes, deques, dictionnaires dans les fonctions

Les listes, deques et dictionnaires sont des objets muables (voir chapitre 1 « Type, listes, boucles, fonctions »).

On considère l'exemple suivant :

```
L1=[2, 3, 5]      # la liste L1 contient les éléments 2, 3, 5
def fct1(L,x):
    L.append(x)   # on ajoute l'élément x à la liste L
                  # inutile d'écrire return(L)
fct1(L1, 8)
print(L1)         # la liste L1 contient les éléments 2, 3, 5, 8
```

Dans Python, les listes, les dictionnaires et les deques sont passés par référence et non par valeur. La variable `L` dans la fonction `fct1` ne contient pas la liste mais fait référence uniquement à l'adresse mémoire de la liste `L1` dans l'ordinateur. Il est donc inutile d'écrire `return (L)` à la fin de la fonction `fct1`.

Cette fonction modifie bien la liste `L1`.

# Énoncés des exercices

## 6.1

■□□□

**Fonction `empiler`**

Proposer une fonction `empiler` qui admet comme argument une pile `P` et un élément `x`. L'élément `x` est inséré en haut de la pile.

## 6.2

■□□□

**Fonction `dépiler`**

Proposer une fonction `dépiler` qui admet comme argument une pile `P` non vide. Cette fonction dépile l'élément situé en haut de la pile et retourne cet élément.

## 6.3

■□□□

**Utilisation des 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`.

**a)** Écrire une fonction `enfiler` qui admet comme arguments une file `F`, un élément `x`. Cette fonction ajoute l'élément x en queue de file.

**b)** 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.

**c)** Écrire une fonction `file_vide` qui admet comme argument une file `F`. Cette fonction retourne `True` si la file est vide et `False` sinon.

**d)** On considère le programme suivant :

```
F=[5, -5, 2]
enfiler(F,7)
print(F)
enfiler(F,8)
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 ?

## 6.4

■□□□

**Utilisation des 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.

**a)** 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.

**b)** 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.

**c)** On considère le programme suivant :

```
L1=[5, 2, -5]
D=deque()          # création d'une deque vide
for elt in L1:
    insere_droite_deque(D, elt)
print(D)
L2=[1, 7, -2]
for elt in L2:
    insere_gauche_deque(D, elt)
print(D)
E=deque([5, 4])
print(len(E)==0)
```

Qu'affiche la console Python lors de l'exécution de ce programme ?

### 6.5

■■□□

**Comptage des éléments d'une liste à l'aide d'un dictionnaire**

On considère la liste `L=[-5.2, 15, 9, 2, 0, 0, 9, 9, -5.2]`.

**a)** Écrire une fonction `comptage` qui admet comme argument une liste. Cette fonction renvoie un dictionnaire permettant de connaître le nombre d'occurrences de chaque élément de la liste.

**b)** Écrire le programme principal permettant d'afficher le nombre d'occurrences de chaque élément de la liste `L`.

## Du mal à démarrer ?

### 6.1

Utiliser `L.append(x)`.

### 6.2

Utiliser `L.pop(x)`.

### 6.3

La liste `F=[5,-5,2]` contient trois éléments. Enfiler l'élément 7 consiste à l'ajouter à la queue de la file (arrière de la file d'attente).

**6.4**

`D.append(x)` permet d'ajouter `x` à l'extrémité droite de la deque `D` comme pour les listes.

`D.appendleft(x)` permet d'ajouter `x` à l'extrémité gauche de la deque `D`. Cette méthode n'est pas correcte avec les listes.

**6.5**

Utiliser une boucle `for` pour parcourir tous les éléments de la liste. Tester si cet élément est présent dans le dictionnaire.

# Corrigés des exercices

**6.1**

La fonction `empiler` ajoute l'élément `x` en haut de la pile en utilisant `P.append(x)`. Il est inutile d'ajouter `return (P)` à la fin de la fonction.

```
def empiler(P,x):
    P.append(x)      # on ajoute l'élément x à la liste P
```

**6.2**

On enlève le dernier élément de la liste en utilisant la méthode `P.pop()`. Il est inutile d'ajouter `return (P)` à la fin de la fonction.

```
def dépiler(P):
    x=P.pop()        # on enlève le dernier élément de la liste P
    return x         # retourne la valeur du dernier élément de la liste P
```

**6.3**

**a)**

```
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
```

**Attention**

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

La liste `F` dans la fonction `enfiler` est un objet muable. Il est donc inutile de retourner `F` dans cette fonction !

**b)**

```
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
```

**c)**

```
def file_vide(F):    # F est une liste Python
                     # la fonction retourne True si la file F est vide
                     # et False sinon
    return F==[]
```

**d)**

Le programme Python affiche :

```
[5, -5, 2, 7]
[5, -5, 2, 7, 8]
F : [-5, 2, 7, 8] x : 5
False
```

## 6.4

**a)**

```
def insere_gauche_deque(D, x):
    D.appendleft(x)  # ajoute x à l'extrémité gauche de la deque D
```

**b)**

```
def insere_droite_deque(D, x):
    D.append(x)      # ajoute x à l'extrémité droite de la deque D
```

**c)**

Le programme Python affiche :

```
deque([5, 2, -5])
deque([-2, 7, 1, 5, 2, -5])
False
```

La deque `E` vaut : `5, 4`. 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('abcd')` définit la deque valant `'a', 'b', 'c', 'd'`.
`D=deque(['abcd'])` définit la deque valant `'abcd'`.
`D=deque([['abcd'], ['ijkl']])` définit la deque valant `['abcd'], ['ijkl']`.
`x=D.popleft()` permet de supprimer `['abcd']` et d'obtenir `x=['abcd']`.

## 6.5

**a)**

```
def comptage(L):
    # la fonction renvoie un dictionnaire permettant de connaître le nombre
    # d'occurrences de chaque élément de la liste
    d={}                    # création d'un dictionnaire vide ou dict()
    for elt in L:
```

```
        if elt in d:
            d[elt]=d[elt]+1  # incrémente de 1 la valeur de la clé elt
        else:
            d[elt]=1         # ajoute la clé elt au dictionnaire
                             # elt apparaît la première fois dans d
                             # la valeur de la clé elt vaut 1
    return d
```

**b)**

```
L=[-5.2, 15, 9, 2, 0, 0, 9, 9, -5.2]
d=comptage(L)
print(d)                     # affichage du dictionnaire
```

Le programme Python affiche : `{-5.2: 2, 15: 1, 9: 3, 2: 1, 0: 2}`.

Cette fonction peut s'appliquer également à une chaîne de caractères.

```
mot="Bonjour"
d=comptage(mot)
print(d)
```

Le programme Python affiche : `{'B': 1, 'o': 2, 'n': 1, 'j': 1, 'u': 1, 'r': 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 `L`. On n'utilise de la place mémoire que pour les caractères qui apparaissent réellement dans `L`.

# Récursivité

CHAPITRE 7

Plan



### Thèmes abordés dans les exercices

- Définition d'une fonction récursive
- Rôle de l'instruction `return`
- Arbre des différents appels d'une fonction récursive
- Méthode « diviser pour régner »

### Points essentiels du cours pour la résolution des exercices

- Identifier une fonction récursive
- Écrire au moins une condition d'arrêt et un appel récursif
- Écrire une version itérative et une version récursive d'une fonction
- Évaluer la complexité d'un algorithme
- Démontrer la terminaison d'un algorithme
- Démontrer la correction d'un algorithme

## Les méthodes à retenir

**Définition**

Une fonction est récursive lorsque le corps de cette fonction fait appel à cette même fonction. Dans le cas contraire, cette fonction est itérative.

### Instruction `return` dans une fonction récursive

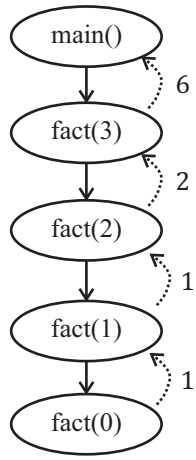
L'instruction `return` doit être présente au moins deux fois dans une fonction récursive : une première fois pour la condition d'arrêt et une deuxième fois pour l'appel récursif.

# Fonction factorielle

On considère la fonction `fact` :

```
def fact(n):
    assert n>=0             # assertion, teste si n>=0
    if n==0:
        return 1            # condition d'arrêt
    else:
        return n*fact(n-1)  # appel récursif
```

L'arbre ci-dessous représente les différents appels de la fonction `fact(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.

Pour calculer la factorielle de $n$, on applique cet algorithme à un sous-problème (ici factorielle de $n$–1). Cette méthode de décomposition/recomposition est appelée « diviser pour régner ».

# Méthode « diviser pour régner »

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 technique de mémoïsation en stockant les résultats déjà calculés (voir chapitre 13 « Programmation dynamique »).

# Terminaison d'un algorithme

Il est essentiel de s'assurer de la terminaison d'une fonction récursive.

Pour démontrer la `terminaison` d'un algorithme, 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.

Exemple de la fonction `fact`. 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 la condition d'arrêt. Le programme se termine donc dans tous les cas si $n \geq 0$. Par contre, le programme ne se termine pas si $n < 0$.

# Correction d'un algorithme

Pour démontrer la correction d'un algorithme, il faut montrer que l'algorithme effectue bien la tâche souhaitée. On prouve très souvent par récurrence pour les algorithmes récursifs qu'un invariant de boucle est vrai.

On considère la propriété (appelée *invariant de boucle*) $P_n$ :

- $P_n$ doit être vraie pour $n = 0$.
- Si $P_n$ est vraie, alors $P_{n+1}$ doit être vraie.

Exemple de la fonction `fact`. Soit $P_n$, « La fonction `fact(n)` retourne $n!$ » :

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

**Remarque**

On peut démontrer simultanément la terminaison et la correction de cet algorithme récursif en considérant la propriété $P_n$, « La fonction `fact(n)` termine et retourne $n!$ ».

# Complexité d'un algorithme

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.

La complexité d'une fonction récursive est souvent plus importante que la complexité de la fonction itérative. Le temps de calcul d'une procédure récursive écrite naïvement peut être très long (voir exercice 7.1 sur la suite de Fibonacci) si l'algorithme calcule plusieurs fois la même valeur.

# Énoncés des exercices

## 7.1

■□□□

**Suite des nombres de Fibonacci**

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

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

**a)** 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 » ?

**b)** Représenter l'arbre des appels de la fonction récursive `fibo1(5)`. Combien de fois est recalculé $F_2$ ? Quel est l'inconvénient ?

## 7.2

■□□□

**Complexité de la fonction factorielle**

Évaluer la complexité de la fonction `fact(n)`.

```
def fact(n):
    assert n>=0        # assertion, teste si n>=0
    if n==0:
        return 1
    else:
        return n*fact(n-1)
```

## 7.3

■■□□

**Algorithme récursif d'Euclide (Maths CCP 2016)**

L'algorithme d'Euclide permet de calculer le pgcd de deux entiers naturels. Voici une fonction Python nommée `euclide` qui implémente l'algorithme d'Euclide :

```
def euclide(a, b):
    U=a
    V=b
    while V!=0:
        r=U%V
        U, V=V, r
    return U
```

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

**b)** Proposer une fonction récursive `euclide_rec` qui calcule le pgcd de deux entiers naturels selon l'algorithme d'Euclide.

## 7.4

**Schéma de Horner**

On cherche à calculer $f(x) = a_0 + a_1 x + ... + a_n x^n$ en utilisant l'algorithme de Horner : $a_n x^n + ... + a_0$ $= \Big(\Big(\big(a_n\big)x + a_{n-1}\Big)x...\Big)x + a_0$ avec $x$ un réel. On définit la liste $L = \big[a_n, a_{n-1}, ..., a_0\big]$.

**a)** Proposer une fonction itérative de l'algorithme de Horner qui admet comme arguments une liste $L$ et un réel $x$ et renvoie $f(x)$. Évaluer la complexité de cette fonction.

**b)** Proposer une version récursive de l'algorithme de Horner. Évaluer la complexité de cette fonction.

**c)** Démontrer la terminaison et la correction de la fonction récursive.

## 7.5

**Recherche récursive par dichotomie d'un élément dans une liste triée**

On considère la liste triée dans l'ordre croissant : `L=[5, 8, 13, 18, 20, 121]`.

**a)** Proposer une fonction récursive `dicho_rec` qui admet comme arguments une liste triée `L` et un élément `x`. Cette fonction retourne `True` si `x` est dans la liste `L`, sinon elle retourne `False`. Elle utilise la méthode de dichotomie.

**b)** Représenter l'arbre des appels de la fonction récursive `dicho_rec(L, 20)`.

## 7.6

**Fonction doublons (Mines Ponts 2017)**

**a)** Proposer une fonction `elim_double(L)` non récursive, de complexité linéaire en la taille de `L`, qui élimine les éléments apparaissant plusieurs fois dans une liste triée `L` et renvoie la liste triée obtenue. Par exemple `elim_double([1, 1, 3, 3, 3, 7])` doit renvoyer la liste [1, 3, 7].

**b)** On dispose de la fonction suivante :

```
def doublons(liste):
    if len(liste)>1:
        if liste[0] != liste[1]:
            return [liste[0]] + doublons(liste[1:])
        del liste[1]
        return doublons(liste)
    else:
        return liste
```

Que retourne l'appel suivant : `doublons([1, 1, 2, 2, 3, 3, 3, 5])` ?

**c)** Représenter l'arbre des appels de la fonction récursive `doublons([1, 2, 2, 5])`.

**d)** Peut-on remplacer `del liste[1]` par `del liste[0]` ?

**e)** Cette fonction est-elle utilisable pour éliminer les éléments apparaissant plusieurs fois dans une liste non triée ? Justifier.

## 7.7

■■□□

**Fonction mystère (Mines Ponts 2019)**

On donne la fonction `mystere` ci-dessous. Le paramètre `x` est un nombre strictement positif et `b` un entier naturel non nul.

```
def mystere(x, b):
    if x < b:
        return 0
    else:
        return 1 + mystere(x / b, b)
```

Que renvoie `mystere(1001,10)` ?

## 7.8

■■■□

**Courbe fractale, flocon de Von Koch**

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

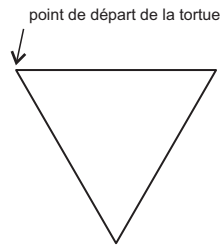
`goto(120,200)` : déplace la tortue au point de coordonnées (120, 200)

`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

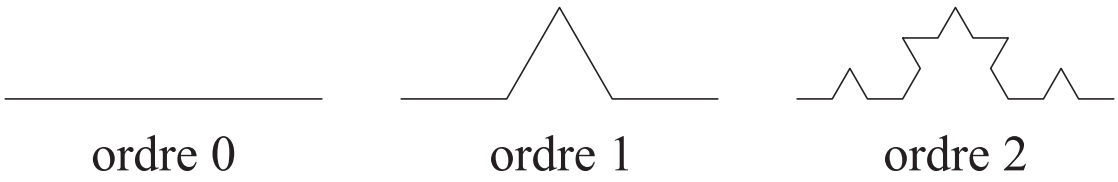
**a)** Écrire une fonction `triangle` qui admet comme argument un entier naturel a et qui trace un triangle équilatéral de côté `a`.




**b)** 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`.

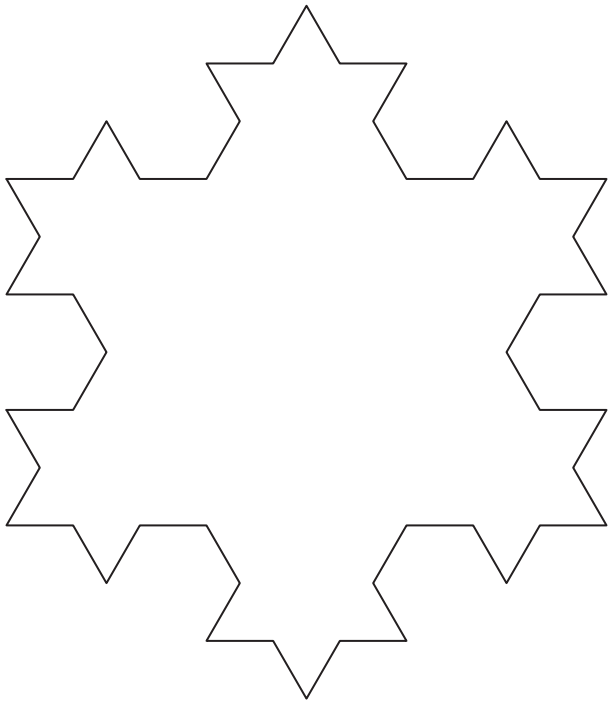
**c)** On souhaite tracer le segment de Koch de longueur *a* à l'ordre *n*.

- 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*.

**d)** 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 suivante 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*.

# Du mal à démarrer ?

**7.1**

**a)** Dans la fonction récursive, écrire deux conditions d'arrêt et un appel récursif.

**7.2**

Définir $T(n)$ le nombre d'opérations élémentaires de la fonction `fact(n)`. Utiliser une récurrence entre $T(n)$ et $T(n-1)$.

### 7.3

**a)** `u%v` calcule le reste de la division euclidienne de $u$ par $v$.

**b)** La condition d'arrêt est `b` = 0.

### 7.4

**a)** Définir une variable `som` et écrire une boucle `for`. À chaque étape, calculer $\text{som} \times x + \text{L}(i)$.

**b)** Utiliser `horner_rec(L[:n-1], x)`.

**c)** Utiliser le variant de boucle `len(L)`. Utiliser une récurrence pour démontrer la correction.

### 7.5

**a)** Calculer l'indice du milieu de la liste `L`. Comparer $x$ à `L[milieu]`. Si $x >$ `L[milieu]`, appeler la fonction récursive à `L[milieu+1:]`.

### 7.6

**a)** Définir une variable $x$ qui prend la première valeur de la liste. Parcourir les autres éléments de la liste.

**b)** La fonction `doublons` est récursive. Elle compare les deux premiers éléments d'une liste.

**c)** Envisager une exécution pas à pas du programme pour écrire l'arbre des appels de la fonction `doublons`.

### 7.7

Utiliser l'arbre des appels de la fonction récursive.

### 7.8

**a)** Utiliser trois fois `forward(a)` et `right(120)`.

**b)** Utiliser `left(360/n)`.

**c)** Utiliser `koch(n-1, a/3)` et `left(60)`.

**d)** Utiliser `koch(n, a)` et `right(120)`.

# Corrigés des exercices

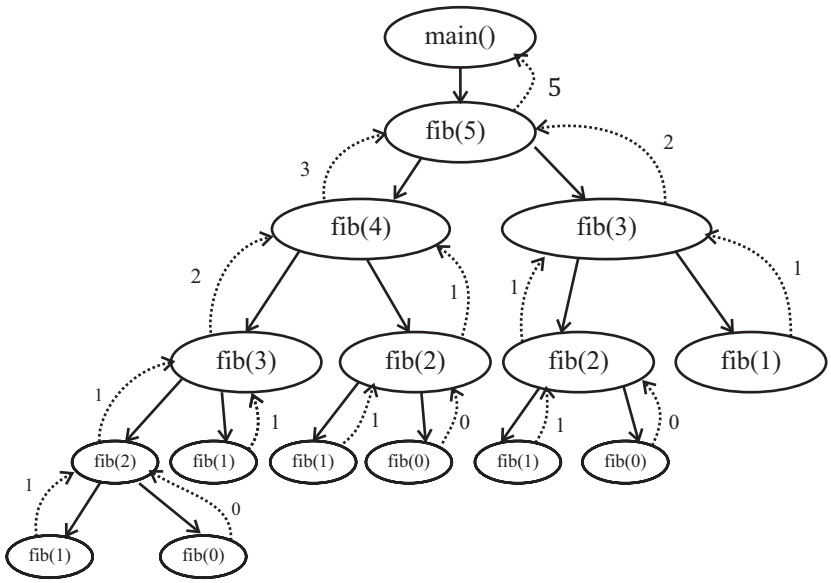
### 7.1

**a)**

```
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.

**b)** 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 $F_n$.

## 7.2

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

On a la récurrence suivante : $T(n) = 2 + T(n-1)$ puisqu'on a deux opérations élémentaires supplémentaires (test, multiplication) à chaque appel de la fonction récursive. On a alors $T(0) = 1$ et $T(n) = 2n + 1$.

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

## 7.3

**a)** On exécute la fonction `euclide(8, 5)`.

- 1er appel de la boucle : $U = 8$ et $V = 5$. Le reste de la division euclidienne de 8 par 5 vaut 3. Le quotient vaut 1. On a alors : $U$=5 et $V$=3.
- 2e appel de la boucle : $U = 5$ et $V = 3$. Le reste de la division euclidienne de 5 par 3 vaut 2. Le quotient vaut 1. On a alors : $U$=3 et $V$=2.
- 3e appel de la boucle : $U = 3$ et $V = 2$. Le reste de la division euclidienne de 3 par 2 vaut 1. Le quotient vaut 1. On a alors : $U = 2$ et $V = 1$.
- 4e appel de la boucle : $U = 2$ et $V = 1$. Le reste de la division euclidienne de 2 par 1 vaut 0. On a alors : $U = 1$ et $V = 0$.

On n'a plus d'appel de la boucle et la fonction retourne 1, qui est le pgcd de 8 et 5.

**b)**

```
def euclide_rec(a, b):
    if b==0:
        return a                        # condition d'arrêt
    else:
        return euclide_rec(b, a%b)      # appel récursif
```

## 7.4

**a)** Pour $n = 2$, on a $a_2x^2 + a_1x + a_0 = \big(\big(a_2\big)x + a_1\big)x + a_0$ avec $L = \big[a_2, a_1, a_0\big]$.

```
def horner(L, x):
    # arguments d'entrée : L(list) et x(float)
    n=len(L)
    som=0
    for i in range(n):
        som=som*x+L[i]
    return som
```

On calcule $T\big(n\big)$ le nombre d'opérations élémentaires de la fonction `horner` :

- 2 opérations élémentaires : affectation de *n* et *som*.
- Pour chaque étape de la boucle `for`, on a 4 opérations élémentaires : appel de l'élément L[i], addition, multiplication et affectation.

Le nombre d'opérations élémentaires vaut 2+4*n*. La complexité de la fonction `horner` est linéaire en $O(n)$.

**b)**

```
def horner_rec(L,x):
    # arguments d'entrée : L(list) et x(float)
    n=len(L)
    if n==1:
        return L[0]                                    # condition d'arrêt
    else:
        return L[len(L)-1]+x*horner_rec(L[:n-1],x)     # appel récursif
```

Le programme récursif fait le contraire de ce que fait le programme itératif, c'est-à-dire que l'on commence par le dernier élément de la liste *L* pour le programme récursif alors que l'on commence par le premier élément de la liste *L* pour le programme itératif.

On calcule $T\big(n\big)$ le nombre d'opérations élémentaires de la fonction `horner_rec` :

- Nombre d'opérations élémentaires avant l'appel récursif : 1 (longueur de *L*) + 1 ($som = 0$) + 1 (test) = 3.
- Récurrence : $T\big(n\big) = 3 + 2\big(\text{multiplication et somme}\big) + T\big(n-1\big)$.

On a donc : $T\big(n\big) = 5 + T\big(n-1\big)$ avec $T\big(1\big) = 2$.

$T\big(2\big) = 5 + 2$, d'où $T\big(n\big) = 5 \times \big(n-1\big) + 2$. La complexité de la fonction `horner_rec` est linéaire en $O(n)$.

**c)** On considère le variant de boucle $n$ = `len(L)`. À chaque appel de la fonction récursive `horner_rec`, il décroît d'une unité (puisqu'on appelle `horner_rec` à $L\big[:n-1\big]$) et finit par atteindre la valeur 1 correspondant à la condition d'arrêt. Le programme se termine donc dans tous les cas si $n \geq 1$.

On considère la propriété (appelée *invariant de boucle*) $P_n$ : « La fonction `horner_rec(n)` retourne $f_n(x)$ ».

- $P_0$ est vrai puisque c'est la condition d'arrêt.
- Supposons que la propriété $P_n$ est vraie.

On pose $f_n(x) = a_0 + a_1x + ... + a_nx^n = \left[\left(\left(\left(a_n\right)x + a_{n-1}\right)x...\right)x + a_0\right]$ associé à la liste $L_n = \left[a_n, a_{n-1}, ..., a_0\right]$.

La fonction `horner_rec(n+1)` réalise l'opération :

$$a'_0 + x\left(f_n(x)\right) = a'_0 + x\left(a_0 + a_1x + ... + a_nx^n\right) = a'_0 + a_0x + a_1x^2 + ... + a_nx^{n+1}$$

On obtient alors : $f_{n+1}(x) = a'_0 + a'_1x + ... + a'_nx^n + a'_{n+1}x^{n+1}$ avec $a'_1 = a_0$, $a'_2 = a_1$,…, $a'_{n+1} = a_n$.

La fonction polynôme $f_{n+1}$ est associée à la liste $L_{n+1} = \left[a'_{n+1}, a'_n, ..., a'_0\right] = \left[a_n, a_{n-1}, ..., a_0\right] + \left[a'_0\right]$.

- Dans la liste $L_n$, $a_0$ est associé au terme $a_0$ dans $f_n(x)$, $a_1$ est associé au terme $a_1x$ dans $f_n(x)$.
- Quand on ajoute un élément $a'_0$ dans la liste $L_n$, $a_0$ est associé à $a_0x$ dans $f_{n+1}(x)$, $a_1$ est associé à $a_1x^2$ dans $f_{n+1}(x)$…

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 `horner_rec` est donc démontrée.

## 7.5

**a)** La méthode de dichotomie 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.

```
def dicho_rec(L, x):
    # la fonction retourne True si x est dans L(list), False sinon
    if L==[]:             # condition d'arrêt si liste vide
        return False
    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 True   # condition d'arrêt si l'élément est trouvé
        elif x>L[milieu]:
                          # recherche entre milieu+1 et la fin de la liste
            return dicho_rec(L[milieu+1:], x)    # appel récursif
        else:
                          # recherche entre début de liste et milieu-1
            return dicho_rec(L[:milieu], x)      # appel récursif
```

On a vu au chapitre 4 « Algorithmes, recherche dans une liste, recherche par dichotomie » une version itérative de la recherche par dichotomie.

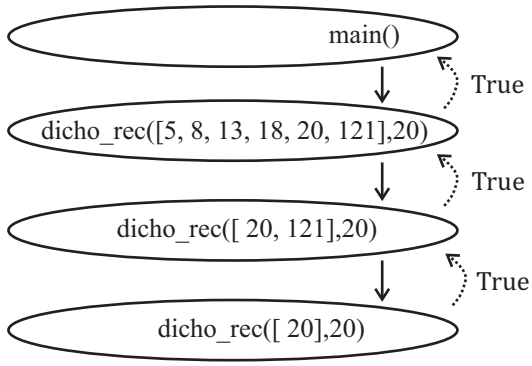
**b)** L'arbre ci-après représente les différents appels de la fonction `dicho_rec(L)` avec `L= [5, 8, 13, 18, 20, 121]`.

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.

- 1er appel de la fonction `dicho_rec([5, 8, 13, 18, 20, 121], 20)` : milieu = 3. On a L[milieu] = 18.

  Comme $x$ > L[milieu], on appelle `dicho_rec([20, 121], 20)`.
- Appel de la fonction `dicho_rec([20, 121],20)` : milieu = 1. On a L[milieu] = 121.

  Comme $x$ < L[milieu], on appelle `dicho_rec([20], 20)`.
- Appel de la fonction `dicho_rec([20], 20)` : milieu = 0. On a L[milieu] = 20.




Comme $x$ = L[milieu], la fonction retourne `True`. La condition d'arrêt est vérifiée. On passe à la phase de remontée.

## 7.6

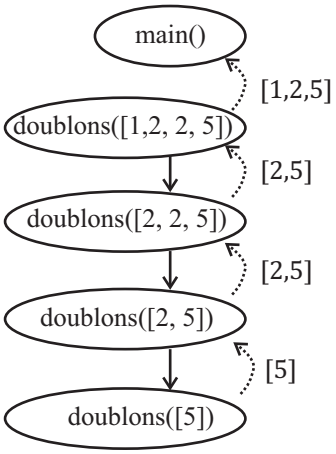
**a)**

```
def elim_double(L):         # argument d'entrée : L(list)
    x=L[0]
    L2=[x]
    n=len(L)
    for i in range(1,n):    # i varie entre 1 inclus et n exclu
        if L[i]!=x:         # si L[i] est différent de x
            x=L[i]          # nouvelle valeur de x pour le prochain test
            L2.append(x)    # on ajoute x à la liste L2
    return L2
```

La complexité de cette fonction est linéaire. On n'a pas de boucle `for` imbriquée.

**b)** La fonction `doublons([1, 1, 2, 2, 3, 3, 3, 5])` retourne [1, 2, 3, 5]. Elle élimine tous les doublons.

**c)** Le programme principal appelle la fonction `doublons([1, 2, 2, 5])`. L'arbre ci-dessous représente les différents appels de la fonction `doublons` :

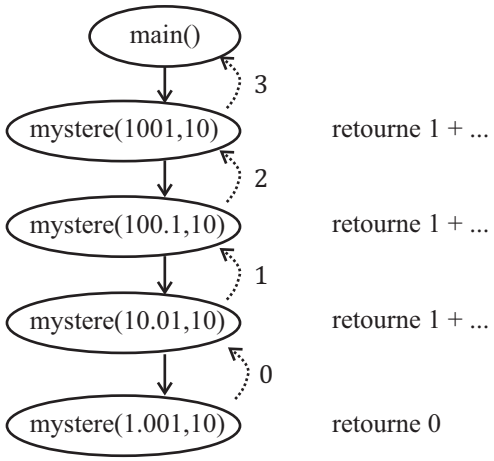



**d)** On peut remplacer `del liste[1]` par `del liste[0]` puisque cette ligne est exécutée uniquement si liste[1] = liste[0]. Si liste[1] ≠ liste[0], l'instruction `return [liste[0]] + doublons(liste[1:])` est exécutée et provoque un arrêt de la fonction.

**e)** On ne peut pas utiliser cette fonction `doublons` avec une liste non triée puisqu'elle compare uniquement deux éléments consécutifs dans une liste. La fonction `doublons([1, 2, 5, 2])` retourne [1, 2, 5, 2] puisqu'elle compare L[1] et L[2]. Elle ne compare pas L[1] et L[3] et n'enlève donc pas L[3].

## 7.7

L'arbre ci-dessous représente les différents appels de la fonction récursive `mystere(1001,10)`.




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.

La fonction `mystere(1001,10)` retourne 3.

**Remarque**

`mystere(3,10)` retourne 0, `mystere(35,10)` retourne 1 et `mystere(352,10)` retourne 2.

## 7.8

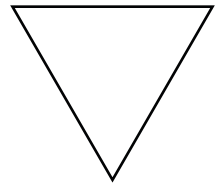
**a)** Pour tracer le triangle équilatéral de côté *a* :

- 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 *
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
```

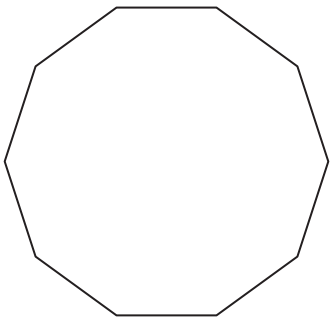
On obtient la figure suivante :



**b)**

```
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
```

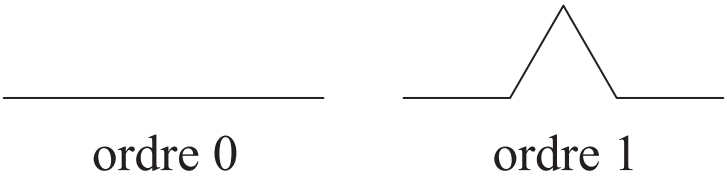
On obtient la figure suivante :



**c)** 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…

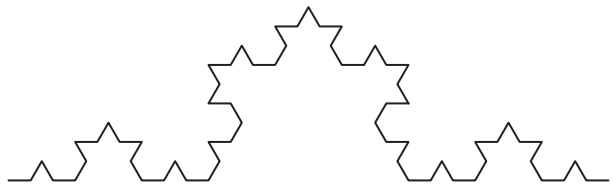
On peut construire le segment de Koch à l'ordre 1 à partir de quatre segments de Koch à l'ordre 0 :

- 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)`.

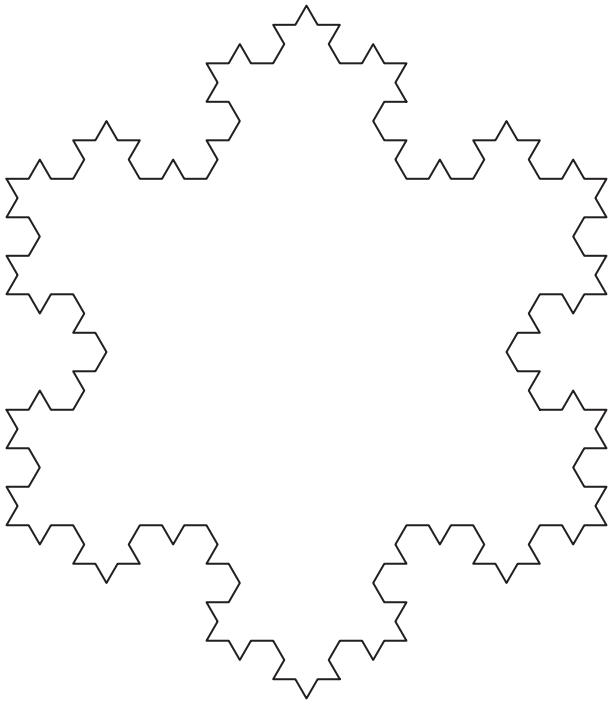
```
def koch(n, a):        # la fonction dessine un segment de Koch
                       # à l'ordre n de longueur a
    if n==0:
        forward(a)     # condition d'arrêt ; avance la tortue de a pixels
    else:              # partage le segment en trois
        # premier tiers : on appelle la fonction 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°
        # troisième 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
```



**d)** 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 :



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.

# Algorithmes gloutons (sauf TSI et TPC)

CHAPITRE 8

Plan



## Thèmes abordés dans les exercices

- Méthode « diviser pour régner »
- Méthode gloutonne
- Faiblesse de la méthode « diviser pour régner »
- Appels récursifs redondants

## Points essentiels du cours pour la résolution des exercices

- Représenter l'arbre des appels récursifs
- Identifier la solution qui semble être la meilleure à chaque étape
- La méthode gloutonne ne donne pas toujours la solution optimale

# Les méthodes à retenir

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.

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 peut s'écrire :

- $L[0] = 0$
- Si $x \geq 1$ : $L[x] = 1 + L\left[x - S[i]\right]$ avec $i$ le plus grand tel que $S[i] \leq x$ .

```
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, 2, 5, 10]
X=11
print(rendu1(S, X))  # on obtient : 2
```

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.

L'algorithme est de type « diviser pour régner » puisqu'on décompose le problème (calcul $L[X]$) en un sous-problème (calcul de $L\left[X - S[i]\right]$ avec $i$ le plus grand tel que $S[i] \leq X$ ). On calcule récursivement le sous-problème.

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).

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.

# Énoncés des exercices

## 8.1

**Problème du sac à dos (sauf TSI et TPC)**

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 entre 0 et $n$–1).
- $S[i][1]$ désigne la masse de l'objet d'indice $i$ notée $m_i$ ($i$ varie entre 0 et $n$–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.

**a)** 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.

**b)** 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 ?

## 8.2

**Découpe d'une barre de métal (sauf TSI et TPC)**

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]]`.

**a)** 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 retourne le prix optimal que l'on peut obtenir de cette barre.

**b)** 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 ?

# Du mal à démarrer ?

## 8.1

**a)** Comme la liste est déjà triée par ordre décroissant valeur/masse, écrire une boucle `for` et tester si on peut ajouter dans le sac à dos l'objet d'indice *i*.

**b)** À l'étape d'indice *i* de la boucle `for`, on insère l'objet qui a le plus grand rapport valeur/masse. On ne revient plus sur le choix par la suite.

## 8.2

**a)** Initialiser à 0 la variable `Ltot`. Comme la liste est déjà triée par ordre décroissant prix/longueur, écrire une boucle `for i in range()` et tester si on coupe la barre à l'abscisse `S[i][1]+Ltot`.

**b)** À chaque étape de de la boucle `for`, on effectue un choix et on ne revient plus sur ce choix.

# Corrigés des exercices

## 8.1

**a)**

```
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*.

**b)** 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 (voir chapitre 13) pour trouver la solution optimale.

## 8.2

**a)** 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 longueur 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.

**b)** 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.

# Matrices de pixels et images

CHAPITRE 9

Plan



### Thèmes abordés dans les exercices

- Images en niveaux de gris
- Inversion de contraste
- Images en couleurs
- Traitement des images
- Masquer une image à l'intérieur d'une autre image

### Points essentiels du cours pour la résolution des exercices

- Transformer une image couleurs en niveaux de gris
- Définir la valeur d'un pixel pour le noir, pour le gris moyen et pour le blanc
- Réaliser un filtre

# Les méthodes à retenir

## Manipulation des images avec Python

Une image est définie par le nombre de pixels. Par exemple, une image 800 × 600 contient 800 pixels en largeur et 600 pixels en hauteur, soit 480 000 pixels. On peut la représenter par une matrice 600 × 800 contenant 600 lignes et 800 colonnes. Le point supérieur gauche de l'image a pour coordonnées [0, 0], le point inférieur droit [599, 799]. On utilise les listes de listes pour représenter les matrices dans Python.

## Utilisation d'une liste de listes

### Matrice

On représente une matrice 2 × 3 par la liste `L` contenant deux listes de longueur 3. Chacune de ces listes de longueur 3 représente une ligne de la matrice $\begin{pmatrix} 3 & 2 & 1 \\ 8 & 6 & 4 \end{pmatrix}$.

```
L=[[3, 2, 1], [8, 6, 4]]
```

Chaque élément de la liste est une liste. Pour extraire le premier élément de la liste `L` :

```
M=L[0]       # M vaut [3, 2, 1]
```

Pour récupérer le deuxième élément de `M` :

```
a=M[1]       # a vaut 2
```

On peut également écrire :

```
b=L[0][1]    # b vaut 2
```

### Bibliothèque PIL

On utilise le module `Image` de la bibliothèque `PIL`. Les instructions permettant d'obtenir la liste `L` seraient rappelées dans un problème de concours :

```
from PIL import Image         # module Image de la bibliothèque PIL
img=Image.open('photo.png')   # stockage des pixels de l'image 'photo.png'
                              # dans la liste img
p,n=img.size                  # n = nombre de lignes   (hauteur)
                              # p = nombre de colonnes (largeur)
L=list(img.getdata())
```

La liste `L` est une simple liste contenant à la suite les pixels de la première ligne de l'image, les pixels de la deuxième ligne…

### Images en niveaux de gris

Chaque élément de la liste `L` est caractérisé par un entier compris entre 0 (noir) et 255 (blanc).

`M` est la matrice qui représente l'ensemble des pixels. La première sous-liste contient la première ligne de l'image. La deuxième sous-liste contient la deuxième ligne de l'image.

```
M=[[0 for j in range(p)] for i in range(n)]
for i in range(n):
    for j in range(p):
        M[i][j]=L[i*p+j]
```

Pour récupérer le pixel à ligne d'indice *i* et à la colonne d'indice *j* :

```
print(M[i][j])    # affiche par exemple 65 : niveau de gris
```

Pour créer une matrice correspondant à une image vide $800 \times 600$ en niveaux de gris :

```
M=[[0 for j in range(800)] for i in range(600)]
```

### Images en couleurs

Chaque élément de la liste `L` est caractérisé par un tuple de 3 valeurs entières comprises entre 0 (intensité nulle) et 255 (intensité maximale) correspondant au codage RVB (rouge, vert, bleu) ou RGB (*red*, *green*, *blue*). On peut représenter $256^3$ couleurs différentes. On rencontre parfois une quatrième valeur correspondant à un coefficient de transparence.

`M` est la matrice qui représente l'ensemble des pixels. La première sous-liste contient la première ligne de l'image. La deuxième sous-liste contient la deuxième ligne de l'image. Chaque pixel de la matrice contient la liste des 3 valeurs entières correspondant au codage RVB.

```
M= [[0 for j in range(p)] for i in range(n)]
for i in range(n):
    for j in range(p):
        M[i][j]=list(L[i*p+j])      # conversion du tuple en liste
```

Pour récupérer le pixel à ligne d'indice *i* et à la colonne d'indice *j* :

```
print(M[i][j])                      # affiche par exemple [85, 80, 96] :
                                    # R, V, B pour une image en couleurs
```

Pour récupérer la valeur du pixel rouge à ligne d'indice *i* et à la colonne d'indice *j* :

```
print(M[i][j][0])                   # affiche 85 : valeur du pixel rouge
```

Chaque couleur d'un pixel prend une valeur entre 0 (intensité nulle) et 255 (intensité maximale). Dans l'exemple ci-dessus, le pixel rouge prend la valeur 85 que l'on peut stocker sur un octet (ou 8 bits).

Pour créer une matrice correspondant à une image vide $800 \times 600$ en couleurs :

```
M=[[[0, 0, 0] for j in range(800)] for i in range(600)]
```

# Énoncés des exercices

## 9.1

### Manipulation d'images

Les instructions suivantes permettent de récupérer dans la liste `L` l'ensemble des pixels d'une image en couleurs. Cette liste contient à la suite les pixels de la première ligne de l'image, de la deuxième ligne… Chaque pixel est caractérisé par un tuple de 3 valeurs entières comprises entre 0 (intensité nulle) et 255 (intensité maximale) correspondant au codage RVB (rouge, vert, bleu). On utilise les listes de listes pour représenter les matrices dans Python.

```
from PIL import Image
img=Image.open('photo.png')  # stockage des pixels de l'image 'photo.png'
                             # dans la liste img
p,n=img.size                 # n = nombre de lignes (hauteur)
                             # p = nombre de colonnes (largeur)
L=list(img.getdata())
```

**a)** Écrire une fonction `creation_couleur_mat` qui admet comme argument l'image en couleurs `img`. Cette fonction retourne la matrice `M` (liste de listes). La première sous-liste contient la première ligne de l'image, la deuxième sous-liste contient la deuxième ligne de l'image… Chaque pixel de la matrice contient la liste des 3 valeurs entières correspondant au codage RVB.

**b)** Écrire une fonction `reduc_image` qui admet comme argument `M` la matrice représentant une image. Cette fonction retourne une nouvelle matrice en ne gardant qu'un pixel sur deux de l'image pour la largeur et la hauteur.

**c)** Écrire une fonction `convertit_min_max` qui admet comme argument `M` la matrice représentant une image en couleurs. Cette fonction convertit une image couleurs en niveaux de gris. La méthode consiste à rechercher le maximum et le minimum pour chaque pixel sur les trois couleurs, puis à faire la moyenne de ces maximums et minimums et ainsi obtenir la valeur du nouveau pixel en niveaux de gris.

**d)** Écrire une fonction `convertit_gris` qui admet comme argument `M` la matrice représentant une image. Cette fonction convertit une image couleurs en niveaux de gris. La méthode consiste à remplacer les trois couleurs de chaque pixel par $Y = 0,299 \times R + 0,587 \times V + 0,114 \times B$.

**e)** Écrire une fonction `inversionconstraste` qui admet comme argument `M` la matrice représentant une image. Cette fonction inverse le contraste d'une image en niveaux de gris. Cette fonction retourne uniquement la moitié supérieure de l'image.

## 9.2

### Filtrage d'images

Les instructions suivantes permettent de récupérer dans la liste `L` l'ensemble des pixels d'une image en niveaux de gris. Cette liste contient à la suite les pixels de la première ligne de l'image, de la deuxième ligne… Les valeurs des pixels sont des entiers qui varient entre 0 (noir) et 255 (blanc). On utilise les listes de listes pour représenter les matrices dans Python.

```
from PIL import Image
img=Image.open('photo.png')  # stockage des pixels de l'image 'photo.png'
                             # dans la liste img
```

```
p,n=img.size                  # n = nombre de lignes (hauteur)
                              # p = nombre de colonnes (largeur)
L=list(img.getdata())
```

**a)** Écrire une fonction `creation_mat` qui admet comme argument l'image `img`. Cette fonction retourne la matrice `M` (liste de listes). La première sous-liste contient la première ligne de l'image, la deuxième sous-liste contient la deuxième ligne de l'image…

**b)** On souhaite réaliser un lissage d'une image représentée par la matrice `M`. On considère la matrice `A` :

| | | |
|---|---|---|
| 1/9 | 1/9 | 1/9 |
| 1/9 | 1/9 | 1/9 |
| 1/9 | 1/9 | 1/9 |

Le traitement suivant est appliqué à la matrice `M`. Pour calculer la nouvelle valeur du pixel de l'image traitée :

- On multiplie son ancienne valeur $\mathbf{M}_{i,j}$ par la valeur centrale de la matrice `A`.
- On additionne les valeurs des pixels adjacents au pixel traité multipliées par les valeurs des éléments adjacents à l'élément central de la matrice `A` : $\mathbf{A}_{0,0} \times \mathbf{M}_{i-1,j-1} + \mathbf{A}_{1,0} \times \mathbf{M}_{i,j-1} + \mathbf{A}_{2,0} \times \mathbf{M}_{i+1,j-1} + ...$
- La nouvelle valeur du pixel est égale à la valeur absolue de la somme précédente.

Écrire une fonction `filtre` qui admet comme arguments `M` la matrice représentant une image et `A` une matrice $3 \times 3$. Cette fonction retourne une nouvelle matrice résultat du filtrage de l'image.

**c)** La dérivée dans la direction horizontale peut être approchée par $\mathbf{M}_{i,j} - \mathbf{M}_{i-1,j}$. Proposer un filtre permettant de détecter le changement d'intensité d'une couleur selon la direction horizontale. On appelle `M3` la matrice de l'image ainsi filtrée.

**d)** Proposer un filtre permettant de détecter le contour selon la direction verticale. On appelle `M4` la matrice de l'image ainsi filtrée.

**e)** Proposer un filtre permettant de détecter les contours dans les deux directions en utilisant pour chaque point de l'image $\sqrt{\left(\mathrm{M3}_{i,j}\right)^2 + \left(\mathrm{M4}_{i,j}\right)^2}$ .

## 9.3

### Stéganographie

Les instructions suivantes permettent de récupérer dans la liste `L` l'ensemble des pixels d'une image en couleurs. Cette liste contient à la suite les pixels de la première ligne de l'image, de la deuxième ligne… Chaque pixel est caractérisé par un tuple de 3 valeurs entières comprises entre 0 (intensité nulle) et 255 (intensité maximale) correspondant au codage RVB (rouge, vert, bleu). On utilise les listes de listes pour représenter les matrices dans Python.

```
from PIL import Image
img=Image.open('photo.png')  # stockage des pixels de l'image 'photo.png'
                             # dans la liste img
p,n=img.size                 # n = nombre de lignes   (hauteur)
                             # p = nombre de colonnes (largeur)
L=list(img.getdata())
```

Le but de la stéganographie est de masquer une image à l'intérieur d'une autre (ou n'importe quel message à l'intérieur d'une image). On considère une image secrète (représentée par la matrice `B`) de même

dimension qu'une image originale (représentée par la matrice `A`). On utilise la méthode LSB : *Least Significant Bit*. Les couleurs des pixels sont stockées sur un octet, on remplace les quatre bits de poids faible (les quatre derniers bits) de l'image originale (matrice `A`) par les quatre bits de poids fort (les quatre premiers bits) du pixel correspondant de l'image secrète (matrice `B`).

**a)** Écrire une fonction `creation_couleur_mat` qui admet comme argument l'image en couleurs `img`. Cette fonction retourne la matrice `M` (liste de listes). La première sous-liste contient la première ligne de l'image, la deuxième sous-liste contient la deuxième ligne de l'image… Chaque pixel de la matrice contient la liste des 3 valeurs entières correspondant au codage RVB.

**b)** Proposer une fonction `bit_poids_faible` qui admet comme argument d'entrée un entier compris entre 0 et 255 et retourne une liste contenant les quatre bits de poids faible.

**c)** Proposer une fonction `bit_poids_fort` qui admet comme argument d'entrée un entier compris entre 0 et 255 et retourne une liste contenant les quatre bits de poids fort.

**d)** Proposer une fonction `entier_poids_fort` qui admet comme argument d'entrée une liste de quatre bits et renvoie un entier compris entre 0 et 255 dont les quatre bits de poids fort sont les quatre bits de l'argument d'entrée. Les quatre bits de poids faible sont nuls.

**e)** Proposer une fonction `creationentier` qui admet comme arguments d'entrée une liste `LA` de quatre bits correspondant à l'image originale et une liste `LB` de quatre bits correspondant à l'image secrète. Cette fonction renvoie un entier dont les quatre bits de poids fort sont `LA` et les quatre bits de poids faible sont `LB`.

**f)** Proposer une fonction `masqueimage(A, B)` qui admet comme arguments d'entrée une image originale (matrice `A`) en couleurs et une image secrète (matrice `B`) en couleurs de même taille. Cette fonction renvoie une matrice contenant à l'intérieur l'image secrète (matrice `B`) en utilisant la méthode LSB.

**g)** Proposer une fonction `recup_image(img)` qui admet comme argument d'entrée une image en couleurs (matrice `M`) et renvoie l'image cachée (matrice `C`) en utilisant la méthode LSB.

# Du mal à démarrer ?

## 9.1

**a)** Utiliser une liste par compréhension pour créer la liste des 3 valeurs correspondant au codage RVB.

**b)** Créer la matrice `M2` par compréhension.

**c)** Effectuer deux boucles `for` pour décrire tous les pixels de l'image.

**e)** `M[i][j][0]` permet d'obtenir la valeur du pixel rouge correspondant à la ligne *i* et la colonne *j*.

**e)** La valeur nulle d'un pixel correspond au noir. La valeur 255 correspond au blanc.

## 9.2

**b)** Effectuer deux boucles `for` pour décrire tous les pixels de l'image.

**c)** Utiliser −1 et 1 dans la matrice `A`.

**d)** La verticale correspond à une colonne dans la matrice `A`.

**e)** Calculer chaque pixel de la nouvelle matrice `M5` en utilisant `M3` et `M4`.

## 9.3

**b)** Utiliser la division euclidienne de l'entier par 2.

**d)** Calculer $2^4 \times$ L[0].

**e)** Calculer $2^4 \times LA[0] + 2^0 \times LB[0]$.

**f)** Utiliser les fonctions `bit_poids_fort` et `creationentier`.

**g)** Utiliser les fonctions `bit_poids_fort` et `bit_poids_faible`.

# Corrigés des exercices

## 9.1

**a)** Il faut convertir le tuple de 3 valeurs en une liste de 3 valeurs. On peut utiliser également `M[i][j]=[elt for elt in L[i*p+j]]` pour convertir le tuple en liste.

```
def creation_couleur_mat(img):
    # la fonction crée une matrice (liste de lignes)
    # à partir de img (image en couleurs)
    p,n=img.size   # n = nombre de lignes   (hauteur)
                   # p = nombre de colonnes (largeur)
    L=list(img.getdata())
    M=[[[0, 0, 0] for j in range(p)] for i in range(n)]
    for i in range(n):
        for j in range(p):
            M[i][j]=list(L[i*p+j]) # pour image couleurs
    return M
```

**b)**

```
def reduc_image(M):
    # la fonction renvoie une matrice M2 (liste de listes)
    # avec 1 pixel sur 2 pour la largeur et la hauteur
    # argument d'entrée : matrice M (liste de listes) en couleurs
    # ou niveaux de gris
    n=len(M)//2               # n = nombre de lignes   (hauteur)
    p=len(M[0])//2            # p = nombre de colonnes (largeur)
    M2=[[[0, 0, 0] for j in range(p)] for i in range(n)]
    for i in range(n):
        for j in range(p):
            M2[i][j]=M[i*2][j*2]
            # si image en niveaux de gris : M2[i][j] entier
            # si image en couleurs : M2[i][j] tuple de 3 valeurs
    return M2
```

**c)**

```
def convertit_min_max(M):
    # la fonction renvoie une matrice M2 (liste de listes)
    # conversion de l'image couleurs en niveaux de gris
    # argument d'entrée : matrice M (liste de listes) en couleurs
    n=len(M)                  # n = nombre de lignes   (hauteur)
    p=len(M[0])               # p = nombre de colonnes (largeur)
```

```
    M2=[[0 for j in range(p)] for i in range(n)]
    for i in range(n):
        for j in range(p):
            a=min(M[i][j])
            b=max(M[i][j])
            M2[i][j]=(a+b)//2 # on obtient un entier
    return M2
```

**d)**

```
def convertit_gris(M):
    # la fonction renvoie une matrice M2 (liste de listes)
    # conversion de l'image couleurs en niveaux de gris
    # argument d'entrée : matrice M (liste de listes) en couleurs
    n=len(M)                  # n = nombre de lignes   (hauteur)
    p=len(M[0])               # p = nombre de colonnes (largeur)
    M2=[[0 for j in range(p)] for i in range(n)]
    for i in range(n):
        for j in range(p):
            M2[i][j]=int(M[i][j][0]*0.299+M[i][j][1]*0.587\
            +M[i][j][2]*0.114)
    return M2
```

**e)** Pour chaque pixel d'une image en niveaux de gris, on a une valeur comprise entre 0 et 255 :

- 0 : couleur noire ;
- 128 : gris moyen ;
- 255 : couleur blanche.

Pour inverser le contraste, on calcule la nouvelle valeur du pixel : `255-img[i,j]`.

```
def inversionconstraste(M):
    # la fonction inverse le contraste de la matrice M (liste de listes)
    # en niveaux de gris. La fonction retourne M2 (liste de listes)
    n=len(M)//2               # n = nombre de lignes   (hauteur)
    p=len(M[0])               # p = nombre de colonnes (largeur)
    M2=[[0 for j in range(p)] for i in range(n)]
    for i in range(n):
        for j in range(p):
            M2[i][j]=255-M[i][j] # inversion du contraste
    return M2
```

**Remarque**

Voir le site Dunod pour télécharger les programmes Python avec des exemples d'images.

## 9.2

**a)**

```
def creation_mat(img):
    # la fonction crée une matrice (liste de lignes)
```

```
    # à partir de img (image en niveaux de gris)
    p,n=img.size   # n = nombre de lignes   (hauteur)
                   # p = nombre de colonnes (largeur)
    L=list(img.getdata())
    M=[[0 for j in range(p)] for i in range(n)]
    for i in range(n):
        for j in range(p):
            M[i][j]=L[i*p+j]
    return M
```

**b)**

```
def somtab(A):              # somme de tous les éléments de
                            # la matrice A (liste de listes)
    n=len(A)                # n = nombre de lignes
    p=len(A[0])             # p = nombre de colonnes
    som=0
    for i in range(n):
        for j in range(p):
            som=som+A[i][j]
    return int(abs(som))    # int pour obtenir un entier

def multipAB(A, B): # multiplication case par case de A et B
    # A et B sont des matrices (listes de listes)
    n=len(A)                # n = nombre de lignes
    p=len(A[0])             # p = nombre de colonnes
    C=[[0 for j in range(p)] for i in range(n)]
    for i in range(n):
        for j in range(p):
           C[i][j]=A[i][j]*B[i][j]
    return C

def filtre(M, A):
    # la fonction renvoie une matrice M2 (liste de listes)
    # filtrage de la matrice M en utilisant la matrice A
    # argument d'entrée : matrice M (liste de listes) en niveaux de gris
    n=len(M)                # n = nombre de lignes   (hauteur)
    p=len(M[0])             # p = nombre de colonnes (largeur)
    M2=[[0 for j in range(p)] for i in range(n)]
    for i in range(1, n-1):
        for j in range(1, p-1):
            B=[[M[i1][j1] for j1 in range(j-1, j+2)] for i1 \
                in range(i-1, i+2)]
            C=multipAB(A, B)
            M2[i][j]=somtab(C)
    return M2

A=[[1/9,1/9,1/9] for i in range(3)]
M2=filtre(M, A)
```

**c)** La dérivée dans la direction horizontale peut être approchée par $\mathbf{M}_{i,j} - \mathbf{M}_{i-1,j}$. La matrice `A` suivante permet de détecter le contour selon la direction horizontale :

| | | |
|---|---|---|
| 0 | 0 | 0 |
| −1 | 1 | 0 |
| 0 | 0 | 0 |

```
A=[[0, 0, 0], [-1, 1, 0], [0, 0, 0]]
M3=filtre(M, A)
```

**d)** La dérivée dans la direction verticale peut être approchée par $\mathbf{M}_{i,j} - \mathbf{M}_{i,j-1}$. La matrice `A` suivante permet de détecter le contour selon la direction verticale :

| | | |
|---|---|---|
| 0 | −1 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

```
A=[[0,-1,0],[0,1,0],[0,0,0]]
M4=filtre(M, A)
```

**e)**

```
def contour(M3, M4):
    # arguments d'entrée : M3 et M4 des matrices (listes de listes)
    # niveaux de gris. La fonction retourne M5 (niveaux de gris)
    # en calculant chaque pixel de M5 à partir de M3 et M4
    import math as m           # module math renommé m
    n=len(M3)                  # n = nombre de lignes   (hauteur)
    p=len(M3[0])               # p = nombre de colonnes (largeur)
    M5=[[0 for j in range(p)] for i in range(n)]
    for i in range(n):
        for j in range(p):
            M5[i][j]=m.sqrt((M3[i][j])**2+(M4[i][j])**2)
    return M5
```

**Remarque**

Voir le site Dunod pour télécharger les programmes Python avec des exemples d'images.

## 9.3

**a)** La fonction `list(L[i*p+j])` permet de convertir le tuple de 3 valeurs (RVB) en une liste de 3 valeurs.

```
def creation_couleur_mat(img):
    # la fonction crée une matrice M (liste de lignes)
    # à partir de img (image en couleurs)
    p,n=img.size   # n = nombre de lignes   (hauteur)
                   # p = nombre de colonnes (largeur)
    L=list(img.getdata())
```

```
    M=[[[0, 0, 0] for j in range(p)] for i in range(n)]
    for i in range(n):
        for j in range(p):
            M[i][j]=list(L[i*p+j]) # pour image couleurs
    return M
```

**b)** Pour coder un entier compris entre 0 et 255, il faut 1 octet, soit 8 bits. La décomposition binaire de l'entier 21 s'écrit : $21 = 10101_b$

On a effet : $21 = 2^4 + 2^2 + 1 = \boxed{1} + 2*(\boxed{0} + 2*(\boxed{1} + 2*(\boxed{0} + 2*(\boxed{1}))))$

On peut effectuer les 4 divisions euclidiennes suivantes :

- 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 $1010\boxed{1}_b$
- 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 $101\boxed{0}1_b$
- Division euclidienne du quotient précédent par 2 : 5//2 = 2 ; 5%2 = $\boxed{1}$
- Division euclidienne du quotient précédent par 2 : 2//2 = 1 ; 2%2 = $\boxed{0}$

On envisage une boucle avec 4 étapes où on effectue les divisions euclidiennes successives pour obtenir les 4 bits de poids faible.

```
def bit_poids_faible(n):
    # n est un entier compris entre 0 et 255
    # la fonction retourne une liste L contenant les 4 bits de poids faible
    L=[]   # liste vide
    for i in range(4):
        r=n%2
        L.append(r)
        n=n//2
    return L
```

`bit_poids_faible(8)` retourne [0, 0, 0, 1].

**c)** On effectue 4 divisions euclidiennes.

À partir de la 5e division euclidienne, on stocke le reste dans une liste `L`.

```
def bit_poids_fort(n):
    # n est un entier compris entre 0 et 255
    # la fonction retourne une liste L contenant les 4 bits de poids fort
    L=[]   # liste vide
    for i in range(8):
        r=n%2
        if i>3:
            L.append(r)
        n=n//2
    return L
```

`bit_poids_fort(128)` retourne [0, 0, 0, 1].

**d)** La fonction `entier_poids_fort(L)` avec `L = [0, 0, 0, 1]` doit retourner l'entier : $n = 2^4 \times L[0] + 2^5 \times L[1] + 2^6 \times L[2] + 2^7 \times L[3] = 128$.

```
def entier_poids_fort(L):
    # L liste de 4 bits (int). La fonction retourne un entier compris
    # entre 0 et 255 dont les 4 bits de poids fort correspondent à L
    n=0
    for i in range(4):
        n+=L[i]*2**(i+4)
    return n
```

**Remarque**

On peut optimiser la fonction en évitant de calculer $2^{i+4}$ à chaque étape de la boucle `for`.

$n = 2^4\left(L[0]+2\left(L[1]+2\left(L[2]+2\left(L[3]\right)\right)\right)\right)$

```
def entier_poids_fort_v2(L):   # deuxième version
    # L liste de 4 bits (int). La fonction retourne un entier compris
    # entre 0 et 255 dont les 4 bits de poids fort correspondent à L
    n=0
    for i in range(3, -1, -1):
                # i varie entre 3 inclus et -1 exclu avec pas = -1
        n=2*n+L[i]
    return n*(2**4)
```

**e)**

```
def creationentier(LA, LB):
    # LA : liste correspondant à l'image originale
    # LB : liste correspondant à l'image secrète
    # la fonction retourne 4 bits de poids fort (LA)
    # et 4 bits de poids faible (LB)
    n=0
    for i in range(4):
        n+=LA[i]*2**(i+4) + LB[i]*2**i
    return n
```

**f)**

```
def masque_image(A, B):
    # image originale : matrice A (liste de listes)
    # image secrète : matrice B (liste de listes)
    # la fonction retourne une matrice M avec méthode LSB
    n=len(A)                  # n = nombre de lignes   (hauteur)
    p=len(A[0])               # p = nombre de colonnes (largeur)
    M=[[[0, 0, 0] for j in range(p)] for i in range(n)]
    for i in range(n):
        for j in range (p):
            for k in range(3):
                LA=bit_poids_fort(A[i][j][k])
                LB=bit_poids_fort(B[i][j][k])
                M[i][j][k]=creationentier(LA, LB)
    return M
```

**g)**

```
def recup_image(M):
    # argument d'entrée : matrice M (liste de listes)
    # la fonction retourne l'image cachée (matrice C) avec méthode LSB
    n=len(M)                  # n = nombre de lignes   (hauteur)
    p=len(M[0])               # p = nombre de colonnes (largeur)
    C=[[[0, 0, 0] for j in range(p)] for i in range(n)]
    for i in range(n):
        for j in range (p):
            for k in range(3):
                L=bit_poids_faible(M[i][j][k])
                C[i][j][k]=entier_poids_fort(L)
    return C
```

**Remarque**

Voir le site Dunod pour télécharger les programmes Python avec des exemples d'images.

# Algorithmes de tri

CHAPITRE 10

Plan



## Thèmes abordés dans les exercices

- Tri par sélection
- Tri par insertion
- Tri par fusion
- Tri rapide
- Tri par comptage, histogramme
- Tri à bulles

## Points essentiels du cours pour la résolution des exercices

- Identifier les différentes méthodes de tri
- Identifier la méthode « diviser pour régner »
- Donner les caractéristiques des tris
- Évaluer la complexité d'un algorithme
- Mise en place d'un algorithme récursif
- Démontrer la correction d'une fonction
- Transformation d'un pseudo-code en programme Python

# Les méthodes à retenir

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.
- 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 que dans la liste non triée.

On rencontre différents algorithmes de tri :

- Tri par sélection : tri comparatif, itératif, stable. Tri en place.
- Tri par insertion : tri comparatif, itératif, stable. Tri en place.
- Tri rapide : tri comparatif, récursif, instable. Tri en place.
- Tri par partition-fusion : tri comparatif, récursif, stable. Le tri n'est pas en place.
- Tri par comptage : tri 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.

# Principe du tri par sélection

Le tri par sélection consiste à chercher le plus petit élément d'une liste, que l'on échange avec le premier. On applique de nouveau cette méthode à la sous-liste restante.

Description étape par étape du tri par sélection :

- Première étape : $i = 0$. On considère la liste `L` non triée : 8 13 1 9 comprenant $n = 4$ éléments. On parcourt la liste du premier au dernier élément pour chercher le minimum de cette liste : 8 13 1 9 (↑ sur 1). On permute alors le minimum (1) avec le premier élément (8) de la liste. On obtient : 1 13 8 9 (↑ sur 1 et 8).
- Deuxième étape : $i = 1$. On cherche le minimum des éléments de `L` dont les indices sont compris entre 1 et $n-1$ : 1 13 8 9 (↑ sur 8). On permute alors le minimum (8) avec le deuxième élément (13) de la liste. On obtient : 1 8 13 9 (↑ sur 8 et 13)
- Troisième étape : $i = 2$. On cherche le minimum des éléments de `L` dont les indices sont compris entre 2 et $n - 1$ : 1 13 8 9 (↑ sur 8). On ne permute pas 8 et 9.

Il est inutile de retourner `L` car la liste est passée par référence.

```
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):
            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=[1, 13, 8, 9]
tri_sélection(L)
```

Le tri par sélection est comparatif et itératif.
On considère la liste `L2=[[213, 'Brésil'], [1444, 'Chine'], [85, 'Iran'],[126, 'Japon'], [111, 'Philippines'], [145, 'Russie'], [85, 'Turquie']]`. On modifie le programme précédent pour trier `L2` par ordre croissant de la population.

```
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):
            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]
```

La liste initiale est triée par ordre alphabétique. 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 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.

La complexité est quadratique en $O\left(n^2\right)$.

# Principe du tri par insertion

Le tri par insertion consiste à insérer les éléments d'une partie de la liste non triée dans la liste triée. Il est souvent utilisé pour trier des cartes.

### Présentation du tri par insertion

On considère la liste L non triée : 8 3 1 9 comprenant $n = 4$ éléments. 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 $\mathrm{L}[k]$ sont déjà triés. Il faut donc insérer cet élément d'indice $k$ dans la liste triée.

Il est inutile de retourner L car la liste est passée par référence.

### Description étape par étape du tri par insertion

- Première étape : $k = 1$. L = 8 3 1 9. On note $x$ l'élément d'indice $k = 1$. Il faut insérer cet élément dans la sous-liste liste triée [8]. Pour cela, on considère une boucle `while`. On part de $i = k$–1. On teste si $x < L[i]$ et $i \geq 0$. Dans ce cas, il faut décaler vers la droite $L[i]$ et on décrémente $i$ d'une unité. Ici, $i$ prend la valeur 0. On obtient alors L = 3 8 1 9.
- Deuxième étape : $k = 2$. L = 3 8 1 9. On note $x$ l'élément d'indice $k = 2$. Il faut insérer cet élément d'indice $k$ dans la sous-liste triée [3, 8]. On considère une boucle `while`. On part de $i = k$–1. On teste si $x < L[i]$ et $i \geq 0$.

Dans ce cas, il faut décaler vers la droite $L[i]$ et on décrémente $i$ d'une unité. Ici, $i$ prend les valeurs : 1 et 0. On obtient alors L = [1][3][8][9].

- Troisième étape : $k = 3$. L = [1][3][8][9]. On note $x$ l'élément d'indice $k = 3$. Il faut insérer cet élément d'indice $k$ dans la sous-liste triée [1, 3, 8]. On considère une boucle `while`. On part de $i = k-1$. Pas de décalage. On obtient alors L = [1][3][8][9].

### Mise en place de l'algorithme

```
def tri_insertion(L):
    n=len(L)
    for k in range(1, n):  # k varie entre 1 inclus et n exclu
        # les éléments entre 0 et k-1 sont triés
        x=L[k]             # on note x l'élément à insérer dans la sous-liste TRIEE
        i=k-1              # on va comparer x et l'élément L[k-1] pour la première exécution
                           # de la boucle while
        while x<L[i] and i>=0:
            L[i+1]=L[i]    # on décale L[i] vers la droite puisque x est situé avant L[i]
            i=i-1          # on décrémente i de 1
        L[i+1]=x
```

Le tri par insertion est comparatif et itératif.

La liste initiale est triée par ordre alphabétique : `[[213, 'Brésil'], [1444, 'Chine'], [85, 'Iran'],[126, 'Japon'], [111, 'Philippines'], [145, 'Russie'], [85, 'Turquie']]`.

La liste triée par ordre croissant de la population est : `[[85, 'Iran'], [85, 'Turquie'], [111, 'Philippines'], [126, 'Japon'], [145, 'Russie'], [213, 'Brésil'], [1444, 'Chine']]`. Le tri par sélection est 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.

Sa complexité spatiale est faible.

La complexité du tri par insertion dans le pire des cas est quadratique en $O(n^2)$.

**Remarque**

Le tri par insertion est très efficace lorsque la liste est déjà presque triée. La complexité est linéaire en $O(n)$.

## Principe du tri par fusion

Le tri par fusion consiste à réunir deux sous-listes triées en une seule liste. Il s'appuie sur la méthode « diviser pour régner ». On partage la liste initiale en deux sous-listes de longueurs quasi égales que l'on trie de façon récursive. Il ne reste plus qu'à fusionner ces deux sous-listes triées.

### Fusion récursive de deux listes triées L1 et L2

On considère deux listes `L1` et `L2` déjà triées.

- Si la liste `L1` est vide, la fusion de `L1` et `L2` est la liste `L2`. La fonction retourne directement `L2`.
- Si la liste `L2` est vide, la fusion de `L1` et `L2` est la liste `L1`. La fonction retourne directement `L1`.

On a deux conditions d'arrêt dans la fonction récursive.

Comme les deux listes `L1` et `L2` sont déjà triées, le premier élément de la fusion de `L1` et `L2` est le plus petit élément entre L1[0] et L2[0]. La fonction `fusion_rec` retourne une liste dont le premier élément est le minimum de L1[0] et L2[0] et dont les autres éléments sont obtenus en appelant `fusion_rec` aux autres éléments de `L1` et `L2`.

```
def fusion_rec(L1, L2):
    if L1==[]:       # condition d'arrêt liste L1 vide
        return L2    # la fusion de L1 et L2 est uniquement L2
    if L2==[]:       # condition d'arrêt liste L2 vide
        return L1    # la fusion de L1 et L2 est uniquement L2
    if L1[0]<L2[0]:
        return [L1[0]]+fusion_rec(L1[1:], L2[:])
        # appel récursif pour ajouter les autres éléments de L1 et L2
    else:
        return [L2[0]]+fusion_rec(L1[:], L2[1:])
```

### Mise en place de l'algorithme récursif

Si la liste contient un seul élément, elle est déjà triée : c'est la condition d'arrêt de la fonction récursive. Sinon, on partage la liste `L` en deux sous-listes `L1` et `L2` que l'on trie récursivement en utilisant la fonction `tri_fusion`. Il ne reste plus qu'à fusionner les deux sous-listes triées.

La complexité du tri par fusion est en $O(n \log n)$.

```
def tri_fusion(L):
    n=len(L)        # nombre d'éléments de la liste L
    if n<=1:
        return L    # condition d'arrêt : la liste est déjà triée
    else:
        milieu=n//2                 # milieu est un entier
        L1=tri_fusion(L[:milieu])   # indices des éléments entre 0 inclus et milieu exclu
        L2=tri_fusion(L[milieu:])   # indices des éléments ≥ milieu
    # on a partagé la liste L en deux sous-listes L1 et L2 triées récursivement
        return fusion_rec(L1, L2)   # fusion des deux sous-listes triées
```

Dans Python, il faut écrire *n*//2 pour que le type du calcul soit *int* et non *float*. L'expression *n*//2 calcule le quotient de la division euclidienne de *n* par 2.

Les caractéristiques du tri par partition-fusion sont : comparatif, récursif, stable. Le tri n'est pas en place.

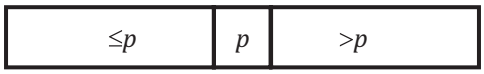
## Principe du tri rapide (sauf TSI et TPC)

Le tri rapide s'appuie sur la méthode « diviser pour régner » comme le tri par fusion.

On choisit dans la liste arbitrairement un pivot *p* (souvent le dernier élément de la liste).

On enlève le pivot *p* de la liste. On segmente la liste en deux sous-listes, l'une contenant les éléments inférieurs ou égaux à *p*, l'autre les éléments restants strictement supérieurs à *p*.

On recommence récursivement avec les listes situées à gauche et à droite du pivot pour rassembler le tout.



### Fonction pivot

On choisit le dernier élément de la liste comme pivot. On pourrait choisir un élément au hasard dans la liste comme pivot.

On réarrange la liste `L` en place en mettant tous les éléments inférieurs ou égaux au pivot à gauche de celui-ci et tous les éléments strictement supérieurs à sa droite.

Le pivot est alors bien placé dans la liste finale à trier.

On définit une fonction `pivot(L, a, b)` qui réarrange la liste `L` comme décrit précédemment. Les deux indices `a` et `b` permettent de caractériser la sous-partie de la liste à trier (indices de début de fin inclus). La fonction `pivot` retourne `ind_p` l'indice du pivot. L'élément `L[ind_piv]` est alors bien placé dans la liste à trier. Il est inutile de retourner `L` car la liste est passée par référence.

```
def pivot(L, a, b):          # indices de début et de fin inclus
    p=L[b]                   # pivot = dernier élément de la liste
    ind_p=a                  # initialisation de l'indice du pivot
    for i in range(a, b):    # i varie entre a inclus et b exclu
        if L[i]<=p:          # teste si élément <= pivot
            L[i], L[ind_p]=L[ind_p], L[i]   # échange des 2 éléments
                                            # (parfois pour rien...)
            ind_p+=1         # on incrémente ind_p de 1
    L[b],L[ind_p]=L[ind_p],L[b]             # échange des 2 éléments
            # inutile de retourner L car passage par référence
            # L[ind_p] est bien placée dans la liste à trier
    return ind_p
```

### Algorithme principal

On définit une fonction récursive `tri_rec(L, a, b)` qui trie les éléments dont les indices sont compris entre `a` et `b` inclus. On appelle la fonction `pivot(L, a, b)`. Cette fonction retourne `ind_p` l'indice du pivot et place le pivot dans la bonne position au sein de la liste à trier.

Il suffit d'appliquer récursivement la fonction `tri_rec` aux éléments situés entre `a` et `b`.

Comme on applique cette fonction récursive à des sous-listes de longueur de plus en plus petite, on arrive à une sous-liste comportant un seul élément qui est forcément triée.

La complexité moyenne du tri rapide est en $O(n \log n)$.

```
def tri_rec(L, a, b):              # a indice de début de liste et b indice de fin de liste
    if b>a:
        ind_p=pivot(L, a, b)
        tri_rec (L, a, ind_p-1)    # tri_rapide pour les indices entre a inclus et ind_p-1 inclus
        tri_rec (L, ind_p+1, b)    # tri rapide pour les indices entre ind_p+1 inclus et b inclus
    # 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 une condition d'arrêt


def tri_rapide(L):
    tri_rec(L,0,len(L)-1)
```

Le tri rapide est comparatif, récursif, instable (il y a parfois un échange des deux éléments qui n'est pas nécessaire dans la fonction pivot). On a un tri en place.

# Énoncés des exercices

## 10.1

■■□□

**Tri par insertion (Mines Ponts 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
```

**a)** 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`.

**b)** 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`.

**c)** 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 ?

**d)** On souhaite, partant d'une liste constituée de couples (chaîne, entier), trier la liste par ordre croissant de l'entier associé suivant le fonctionnement suivant :

```
L=[['Bresil', 76], ['Kenya', 26017], ['Ouganda', 8431]]
tri_chaine(L)
print(L)
```

On obtient : `[['Bresil', 76], ['Ouganda', 8431], ['Kenya', 26017]]`

Écrire une fonction `tri_chaine` réalisant cette opération.

## 10.2

■□□□

**Tri par sélection (Banque PT 2018)**

On considère une liste de listes nommée `T`, dont chaque ligne `T[i]` est construite de la façon suivante :

[nom_coureur, prenom_coureur, nom_epreuve, date_epreuve,temps]

où chaque élément est une chaîne, à l'exception de `temps` qui est un flottant.

L'entraîneur souhaite obtenir des données sur les dix meilleurs temps d'un 100 mètres, réalisés par les membres de l'équipe (on suppose que `T` possède au moins 10 lignes. On adapte l'algorithme de tri suivant, dit du *tri par sélection*, présenté ici dans le cas d'une liste simple `L` que l'on trie par ordre croissant :

1) On recherche le plus petit élément de `L` et on l'échange avec le premier élément.

2) On recherche le deuxième plus petit élément de `L` (qui est donc le plus petit élément de `L[1 :]` et on l'échange avec le deuxième élément.

3) On recherche le troisième plus petit élément de `L` (qui est donc le plus petit élément de `L[2 :]` et on l'échange avec le troisième élément, et ainsi de suite jusqu'à ce que la liste soit entièrement triée.

**a)** Proposer une fonction `top10(T)` prenant comme argument la liste `T` et retournant une liste au même format, mais limitée aux données relatives aux 10 meilleurs temps classés par ordre croissant.

**b)** En justifiant la réponse, donner la complexité de la fonction `top10` en fonction du nombre *N* de performances évaluées dans la liste `T`.

**c)** Si l'on suppose *N* très grand, quel est l'intérêt de la technique utilisée ici par rapport à l'approche « naïve » consistant à trier entièrement la liste grâce à un algorithme performant (tri rapide, tri par fusion…) puis à en extraire les 10 meilleurs temps ?

**d)** Le tri par sélection serait-il toujours avantageux si l'on souhaitait trier entièrement `T` par temps croissants au lieu de se limiter aux 10 meilleurs temps ? Pourquoi ?

**e)** Donner les caractéristiques du tri par sélection.

## 10.3

■■□□

**Fonction congestion (CCP PSI 2017)**

On considère une station de mesure permettant d'étudier la congestion d'une autoroute. Chaque point de mesure représente une tranche de temps de 6 minutes. On obtient une liste des débits `q_exp`, représentatif du nombre de véhicules par unité de temps, et une liste des vitesses moyennes `v_exp`. Au niveau d'une station de mesure, la situation est dite congestionnée lorsque les vitesses prises par les véhicules restent inférieures à 40 km/h et la situation est dite fluide lorsque les vitesses restent supérieures à 80 km/h.

**a)** La fonction `congestion(v_exp)`, qui prend en argument `v_exp` et renvoie la valeur médiane de la liste de valeurs `v_exp` est définie ci-dessous. La recherche de la médiane est basée sur un algorithme de tri. Choisir une des 4 propositions données pour compléter les 2 lignes manquantes (indiquées par « `ligne à compléter` »). La longueur de `v_exp` est impaire.

```
def congestion(v_exp):
    nbmesures=len(v_exp)
    for i in range(nbmesures):
        v=v_exp[i]
        j = i
        while 0 < j and v < v_exp[j-1]:
                # ligne à compléter
                # ligne à compléter
        v_exp[j] = v
    return v_exp[nbmesures//2]
```

Propositions pour les lignes manquantes :

| | |
|---|---|
| Proposition n° 1 | `v_exp[j-1] = v_exp[j]`<br>`j = j-1` |
| Proposition n° 2 | `v_exp[j] = v_exp[j-1]`<br>`j = j-1` |
| Proposition n° 3 | `v_exp[j+1] = v_exp[j]`<br>`j = j+1` |
| Proposition n° 4 | `v_exp[j] = v_exp[j+1]`<br>`j = j+1` |

**b)** Donner le nom puis la complexité de l'algorithme de tri employé, dans le meilleur et dans le pire des cas.

**c)** On exécute la fonction `congestion(v_exp)`. La valeur retournée par la fonction étant 30, quelle conclusion peut-on tirer de ce résultat ?

## 10.4

■■■□

**Fonction médiane (CCP PSI 2015) (sauf TSI et TPC)**

**a)** Proposer une fonction `mediane(L)`, basée sur un algorithme de tri par fusion, qui renvoie la valeur médiane d'une liste de valeurs `L`.

**b)** Donner sans justification sa complexité.

## 10.5

■■■■

**Tri par insertion et tri rapide (Mines Ponts 2018) (sauf TSI et TPC)**

On cherche à trier la liste des propriétés des vagues. La méthode utilisée est un tri rapide (*quick sort*). On donne ci-dessous un algorithme possible pour la fonction `triRapide`. Trois arguments sont nécessaires : une liste `liste`, et deux indices `g` et `d`.

```
1  def triRapide(liste, g, d):
2      pivot
3      i=g
4      j=d
5      while True:
6          while i<=d and liste[i][0]<pivot:
7              i=i+1
8          while j>=g and liste[j][0]>pivot:
9              j=j-1
10         if i>j:
11             break
12         if i<j:
13             liste[i], liste[j]=liste[j], liste[i]
14         i=i+1
15         j=j-1
16     if g<j:
17         triRapide(liste, g, j)
18     if i<d:
19         triRapide(liste, i, d)
```

**a)** Préciser les valeurs que doivent prendre les arguments `g` et `d` au premier appel de la fonction `triRapide`. Compléter la ligne 2.

Lorsque le tri rapide est utilisé et que le nombre de données à traiter devient petit dans les sous-listes (de l'ordre de 15), il peut être avantageux d'utiliser un « tri par insertion ». On appelle `triInsertion` la fonction qui permet d'effectuer un tri par insertion. Elle admet en argument une liste `liste`, et deux indices `g` et `g`. Ces deux indices permettent de caractériser la sous-partie de la liste à trier (indices de début et de fin inclus).

**b)** Donner les modifications à apporter à la fonction `triRapide` pour que, lorsque le nombre de données dans une sous-liste devient inférieur ou égal à 15, la fonction `triInsertion` soit appelée pour déterminer le tri.

Le code incomplet de la fonction `triInsertion` est donné ci-dessous.

**c)** La fonction `triInsertion` admet trois arguments : une liste de données `liste`, et deux indices `g` et `d`. Elle trie dans l'ordre croissant la partie de la liste comprise entre les indices `g` et `d` inclus. Compléter cette fonction avec le nombre de lignes de votre choix.

```
def triInsertion(liste, g, d):
    for i in range(g+1, d+1):
        j=i-1
        tmp=liste[i]
        while    # à compléter




        liste[j+1]=tmp
```

## 10.6

**Transformation d'un pseudo-code en programme Python (sauf TSI et TPC)**

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

**a)** Écrire le programme Python correspondant à ce pseudo-code.

**b)** Écrire le programme permettant de trier la liste `L=[10, 3, 9, 6, 8]`. Comparer la fonction `quicksort` avec la fonction `tri_rapide` (voir rappel de cours « Principe du tri rapide »). Que vaut la liste `L` en sortie de programme ? Représenter l'arbre des appels de la fonction récursive `quicksort`.

**c)** Proposer une amélioration du programme pour ne pas modifier la liste initiale `L` en sortie de programme.

## 10.7

■■□□

**Tri par comptage, histogramme (Mines Ponts 2016)**

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 une liste `HISTO` de $p + 1$ valeurs initialisées à 0.
- On parcourt la liste `L`, on compte le nombre de fois qu'apparaît `L[i]` et on incrémente `HISTO[L[i]]` de 1 à chaque fois.
- On parcourt la liste `HISTO` pour construire au fur et à mesure la liste triée `L_tri`.

**a)** Écrire une fonction `tri_comptage` réalisant ce tri.

**b)** Donner les caractéristiques du tri par comptage.

**c)** Évaluer la complexité dans le pire des cas lors de l'appel de la fonction `tri_comptage` en fonction de $n$ et $p$.

**d)** 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 ».

**e)** Proposer une amélioration de la fonction `tri_comptage` en tenant compte du minimum et du maximum de la liste `L`.

## 10.8

■■■□

**Tri à bulles (sauf TSI et TPC)**

On considère la liste `L=[8, 4, 2, 22, 6]` composée de $n$ éléments. 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… Au cours d'une passe de la liste, les plus grands éléments remontent de proche en proche vers la droite comme des bulles vers la surface.

On réitère l'opération précédente en s'arrêtant à l'élément d'indice $n - 2$ puis à l'élément d'indice $n - 3$…

**a)** Écrire une fonction `tri_bulles` réalisant cette opération. Décrire les différentes étapes du tri à bulles appliqué à la liste `L`.

**b)** On considère la liste triée par ordre alphabétique de la population en millions d'habitants :

```
P= [[213, 'Brésil'], [1444, 'Chine'], [85, 'Iran'],[126, 'Japon'],
[111, 'Philippines'], [145, 'Russie'], [85, 'Turquie']]
```

Écrire une fonction `tri_bulles2` permettant de trier par ordre croissant de la population la liste `P`. Donner les caractéristiques du tri à bulles.

**c)** Évaluer la complexité dans le pire des cas lors de l'appel de la fonction `tri_bulles`.

**d)** 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.

# Du mal à démarrer ?

## 10.1

**a)** Bien analyser chaque étape de la boucle `for`.

**b)** Pour démontrer la correction d'un programme, on utilise une propriété appelée **invariant de boucle** qui doit avoir trois caractéristiques : 1) être vérifiée avant d'entrer en boucle ; 2) si elle est vérifiée avant une itération, alors elle doit être vérifiée après cette itération ; 3) sa vérification en fin de boucle permet d'en déduire que le programme est correct.

**c)** Définir $T(n)$ le nombre d'opérations élémentaires de la fonction `tri(L)`. Compter le nombre d'opérations élémentaires pour chaque étape de la boucle `for`.

**d)** `L` est une liste de listes. On peut écrire `x=L[i][0]` ; `y=L[i][1]` ou `x,y=L[i]`.

## 10.2

**a)** Définir une première boucle avec 10 étapes. Définir une autre boucle imbriquée à la première.

**b)** Calculer un ordre de grandeur du nombre d'opérations élémentaires pour chaque étape de la première boucle `for`.

**c)** La liste `T` n'est pas triée entièrement.

**d)** Calculer un ordre de grandeur du nombre d'opérations élémentaires.

## 10.3

**a)** Il s'agit d'un tri par insertion.

**b)** Définir $T(n)$ le nombre d'opérations élémentaires de la fonction `tri(L)`. Compter le nombre d'opérations élémentaires pour chaque étape de la boucle `for`.

## 10.4

**a)** Trier la liste `L`. Considérer deux cas selon la parité du nombre d'éléments de la liste `L`.

**10.5**

**a)** Il s'agit d'un tri rapide. Choisir par exemple le dernier élément de la liste comme pivot.

**b)** Faire le test si $d - g$ est inférieur ou égal à 15.

**c)** L'élément `tmp` est à insérer dans la sous-liste triée entre $g$ et $i$–1. Définir une bouche `while` en décrémentant $j$ de 1.

**10.6**

**a)** La condition d'arrêt de la fonction récursive est `len(L)<=1`. Si cette condition n'est pas vérifiée, enlever le dernier élément de `L`. Parcourir tous les éléments restants de `L`. Si `x <= p`, ajouter `x` à la liste `L1`.

**c)** On ne modifie pas la liste `L` initiale si on enlève la ligne `L.pop()`. Modifier la boucle `for` pour parcourir tous les éléments de `L` sauf le dernier.

**10.7**

**a)** Définir une liste `HISTO`. Parcourir la liste `L` pour incrémenter les éléments de la liste `HISTO`.

**c)** Compter le nombre d'opérations élémentaires.

**e)** Calculer `mini` le minimum des éléments de la liste `L` et `maxi` le maximum des éléments de la liste `L`. `HISTO[0]` correspond à `mini` et non à 0. Parcourir les éléments de `L` et incrémenter de 1 : `HISTO[L[i]-mini]+=1`.

**10.8**

**a)** On parcourt la liste `L` en comparant les éléments consécutifs deux à deux 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 ».

**b)** Il faut comparer `L[j][0]` à `L[j-1][0]`.

**c)** Compter le nombre d'opérations élémentaires.

**d)** Définir une variable permettant de savoir si des éléments ont été échangés lors d'un passage.

# Corrigés des exercices

**10.1**

**a)** On considère la liste `L=[5, 2, 3, 1, 4]`. Le nombre d'éléments est $n = 5$.

- 1re itération : `L=[2, 5, 3, 1, 4]`
- 2e itération : `L=[2, 3, 5, 1, 4]`
- 3e itération : `L=[1, 2, 3, 5, 4]`
- 4e itération : `L=[1, 2, 3, 4, 5]`

On obtient la liste triée en place.

**b)** On considère la propriété (appelée invariant de boucle) $P_i$ : « la liste `L[0:i+1]` est triée par ordre croissant à l'issue de l'itération $i$ ».

- $i = 0$. La liste contient un élément. On a donc une liste triée.

**Attention**

`L[0:i+1]` permet de faire une extraction de la liste `L`.

- 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.

On a démontré la correction de la fonction `tri`.

**c)** On définit $T(n)$ le nombre d'opérations élémentaires de la fonction `tri(L)`.

- Ligne 2 : deux 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 : une opération élémentaire (affectation) ;
  - ligne 5 : deux opérations élémentaires (affectation et appel de l'élément `L[i]`) ;
  - ligne 6 : quatre 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`) : cinq opérations élémentaires (appel de l'élément `L[j-1]`, appel de l'élément `L[j]`, affectation, calcul `j-1`, affectation).
- Ligne 9 : deux opérations élémentaires (appel de l'élément `L[j]` et affectation).

Dans le meilleur des cas, le nombre total d'opérations élémentaires est :

$$T(n) = 2 + \sum_{i=1}^{n-1} 9 = 2 + 9(n-1) = 9n - 7$$

La complexité dans le meilleur des cas est linéaire en $O(n)$.

Dans le pire des cas, le nombre total d'opérations élémentaires est :

$$T(n) = 2 + \sum_{i=1}^{n-1} (9 + 5i) = 2 + 9(n-1) + 5\frac{(n-1)n}{2}$$

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 en $O(n \log n)$.

**d)**

```
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]
```

## 10.2

**a)**

```
def top10(T):    # argument d'entrée : T(liste de listes)
    N=len(T)
    for i in range(10):
        min=T[i][4]
        ind_min=i
        for j in range(i+1, N):
            if T[j][4]<min:
                min=T[j][4]    # nouvelle valeur du minimum
                ind_min=j      # indice du minimum
        if ind_min>i:
            T[i],T[ind_min]=T[ind_min],T[i]
            # échange entre T[i] et T[ind_min]
    return T[0:10]
```

**b)** On définit $T(N)$ le nombre d'opérations élémentaires dans le pire des cas de la fonction `top10`. On calcule le nombre d'opérations élémentaires dans le pire des cas.

La boucle `for i in range(10)` est parcourue 10 fois. À chaque étape, on a :

- Trois opérations élémentaires : appel de l'élément `T[i][4]` et deux affectations.
- À chaque étape de la boucle *j,* on a cinq opérations élémentaires (appel d'un élément, comparaison, appel d'un élément, deux affectations). On a donc pour toutes les étapes de la boucle $j$ : $5 \times (10 - (i+1)) = 5((9-i))$ opérations élémentaires.
- Une comparaison, deux appels et deux affectations.

Au total, on a $\sum_{i=0}^{9}\left(3+\sum_{j=i+1}^{N-1}5(9-i)+5\right)$ opérations élémentaires. Le calcul exact donne $-745+225N$.

On peut faire un calcul approché. La boucle *j* est exécutée *N* fois. On a donc 5*N* opérations élémentaires pour les étapes de la boucle *j*. La boucle *i* est exécutée 10 fois. On a donc en ordre de grandeur : $10\times(3+5N+5)$ opérations élémentaires.

La complexité est linéaire en $O(N)$.

**c)** La complexité du tri par insertion est quadratique en $O(N^2)$. La complexité du tri rapide et du tri par fusion est en $O(N\log N)$. Le tri par sélection est bien plus efficace si *N* est très grand puisqu'on ne trie pas la liste `T` entièrement. La boucle `for i in range` n'est exécutée que 10 fois.

**d)** Dans ce cas, la boucle *i* est exécutée *N* fois. Pour chaque valeur de *i*, la boucle *j* est exécutée *N* fois en ordre de grandeur. On a donc une complexité en $O(N^2)$. Les algorithmes de tri rapide ou par fusion sont beaucoup plus avantageux.

**e)** Le tri par sélection est comparatif et itératif.

La liste initiale est triée par ordre alphabétique **:** `[[213, 'Brésil'], [1444, 'Chine'], [85, 'Iran'],[126, 'Japon'], [111, 'Philippines'], [145, 'Russie'], [85, 'Turquie']]`.

La liste triée par ordre croissant de la population est : `[[85, 'Iran'], [85, 'Turquie'], [111, 'Philippines'], [126, 'Japon'], [145, 'Russie'], [213, 'Brésil'], [1444, 'Chine']]`. Le tri par sélection est 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.

## 10.3

**a)** On réalise un tri par insertion.

```
def congestion(v_exp):
    nbmesures=len(v_exp)
    for i in range(nbmesures):
        v=v_exp[i]
        j = i
        while 0 < j and v < v_exp[j-1]:
            v_exp[j]=v_exp[j-1]  # on décale v_exp[j-1] vers la droite
            j=j-1                # on décrémente j de 1
        v_exp[j] = v
    return v_exp, v_exp[nbmesures//2]
```

La proposition correcte est la proposition n° 2. Si $i>1$, on considère que les éléments entre 0 et $i-1$ sont triés. On note `v = v_exp[i]` l'élément à insérer dans la sous-liste triée. Si `v < v_exp[j-1]`, alors on décale `v_exp[j-1]` vers la droite. Il faut ensuite décrémenter *j* de 1 pour décaler éventuellement les autres éléments.

**b)** On note *n* le nombre d'éléments de la liste `v_exp`.

On calcule $T(n)$ le nombre d'opérations élémentaires de la fonction `congestion(v_exp)`:

- Deux opérations élémentaires (affectation et fonction `len()`).
- La variable *i* prend toutes les valeurs de 0 à $n-1$ et, pour une valeur *i* fixée, on a :
  - deux opérations élémentaires (affectation et appel de l'élément `v_exp[i]`) ;

- une affectation ($j = i$) ;
- ligne `while` : quatre opérations élémentaires (deux tests, opération `and`, appel de l'élément `v_exp[j-1]`).

  Passage dans la boucle `while` :

  – 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`) : cinq opérations élémentaires (appel de l'élément `v_exp[j-1]`, appel de l'élément `v_exp[j]`, affectation, calcul de $j-1$, affectation) ;
- deux opérations élémentaires (appel de l'élément `v_exp[j]` et affectation).

Dans le meilleur des cas, le nombre total d'opérations élémentaires est :

$$2+\sum_{i=0}^{n-1} 9 = 2+9n$$

La complexité dans le meilleur des cas est linéaire en $O(n)$.

Dans le pire des cas, le nombre total d'opérations élémentaires est :

$$2+\sum_{i=0}^{n-1} (9+5i) = 2+9n+5\frac{(n-1)n}{2}$$

La complexité dans le pire des cas est quadratique en $O(n^2)$.

**c)** La fonction `congestion(v_exp)` retourne la valeur médiane de la liste `v_exp`. Pendant plus de la moitié du temps d'observation, les véhicules ont une vitesse moyenne inférieure à 30 km/h, la circulation est donc congestionnée.

## 10.4

**a)** On a étudié le tri par fusion dans les rappels de cours. Pour calculer la médiane d'une liste de valeurs, il faut d'abord trier cette liste de valeurs. On considère deux cas selon la parité de $n$ (nombre d'éléments de `L`). On note `L_tri` la liste triée.

- Si $n$ est impair, la médiane est la valeur de `L_tri` $\left[\frac{n-1}{2}\right]$. Par exemple, la médiane de $[2,3,8]$ est 2 avec $n = 3$. Dans Python, il faut écrire `(n-1)//2` pour que le type du calcul soit `int` et non `float`. L'expression `(n-1)//2` calcule le quotient de la division euclidienne de $n-1$ par 2.
- Si $n$ est pair, la médiane est la moyenne de `L_tri` $\left[\frac{n}{2}-1\right]$ et `L_tri` $\left[\frac{n}{2}\right]$. Par exemple, la médiane de [2, 3, 4, 8] est 3,5 avec $n = 4$.

```
def fusion_rec(L1, L2):      # arguments d'entrée : listes L1 et L2
    if L1==[]:       # condition d'arrêt liste L1 vide
        return L2    # la fusion de L1 et L2 est uniquement L2
    if L2==[]:       # condition d'arrêt liste L2 vide
        return L1    # la fusion de L1 et L2 est uniquement L2
    if L1[0]<L2[0]:
                     # les deux listes L1 et L2 sont déjà triées
                     # le premier élément de la fusion de L1 et L2
                     # est le plus petit entre L1[0] et L2[0]
        return [L1[0]]+fusion_rec(L1[1:],L2[:])
        # appel récursif pour ajouter les autres éléments de L1 et L2
```

```
    else:
        return [L2[0]]+fusion_rec(L1[:], L2[1:])

def tri_fusion(L):                  # argument d'entrée : liste L
    n=len(L)
    if n<=1:
        return L                    # condition d'arrêt : la liste est déjà triée
    else:
        milieu=n//2
        L1=tri_fusion(L[:milieu])   # indices des éléments entre 0 inclus
                                    # et milieu exclu
        L2=tri_fusion(L[milieu:])   # indices des éléments >= milieu
        return fusion_rec(L1, L2)   # fusion des deux sous-listes triées

def mediane(L):                     # argument d'entrée : liste L
    Ltri=tri_fusion(L)
    n=len(L)
    if n%2==0:
        # nombre pair d'éléments dans la liste
        return (Ltri[n//2]+Ltri[n//2-1])/2
    else:
        # nombre impair d'éléments dans la liste
        return Ltri[(n-1)//2]
```

**b)** La complexité du tri par fusion est en $O(n \log n)$.

## 10.5

**a)** Au premier appel de la fonction `triRapide`, on prend `g` = 0 et `d` = `len(liste) - 1`.

Les deux indices `g` et `d` permettent de caractériser la sous-partie de la liste à trier (indices de début et de fin inclus).

`pivot=[d][0]` : le pivot est le dernier élément de la liste. On peut choisir un élément au hasard dans la liste comme pivot.

**b)**

```
def triRapide(liste, g, d):
    if (d-g)<=15:
        triInsertion(liste, g, d)
    else:    # le reste du programme n'est pas modifié
        pivot=liste[d][0]
        i=g
        j=d
        while True:
            while i<=d and liste[i][0]<pivot:
                i=i+1
            while j>=g and liste[j][0]>pivot:
                j=j-1
            if i>j:
```

```
                break
            if i<j:
                liste[i],liste[j]=liste[j],liste[i]
            i=i+1
            j=j-1
        if g<j:
            triRapide(liste, g, j)
        if i<d:
            triRapide(liste, i, d)
```

**c)**

```
def triInsertion(liste, g, d):
    for i in range(g+1, d+1):    # i varie entre g+1 et d inclus
        j=i-1                    # les éléments entre g et i-1 sont triés
        tmp=liste[i]             # tmp à insérer dans la sous-liste triée entre g
                                 # et i-1
        while j>=g and tmp<liste[j]:
            liste[j+1]=liste[j]  # on décale liste[j] vers la droite
            j=j-1                # on décrémente j de 1
        liste[j+1]=tmp
```

## 10.6

**a)**

```
def quicksort(L):
    # la fonction retourne L3 la liste triée par ordre croissant
    # de la liste 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
        L.pop()           # enlève le dernier élément de L
        L1, L2=[], []     # création de deux listes vides
        for x in L:       # parcourt les éléments de la liste L
            if x<=p:
                L1+=[x]   # ajoute x à L1
            else:
                L2+=[x]   # ajoute x à L2
    L3=quicksort(L1)+[p]+quicksort(L2)   # concaténation de 3 listes
    return L3
```

**b)**

```
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]
```

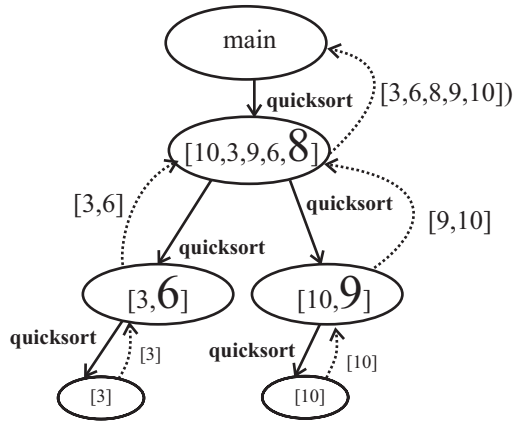
La fonction `quicksort` fonctionne de la même façon que la fonction `tri_rapide` du rappel de cours « Principe du tri rapide ». On choisit le dernier élément de la liste qui est appelé le pivot. On place dans la liste `L1` tous les éléments inférieurs ou égaux au pivot et dans la liste `L2` tous les éléments strictement supérieurs au pivot. On crée une liste `L3` qui est la même que celle obtenue à la fin de la fonction `pivot` du rappel de cours « Principe du 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]`.

L'arbre ci-dessous représente les différents appels de la fonction `quicksort`. 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]`.

**c)** 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):
    # la fonction retourne L3 la liste triée par ordre croissant de la liste 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
        L1, L2=[], []     # création de deux listes vides
        for i in range(0, n-1): # parcourt tous les éléments
                                # de la liste L sauf le dernier
```

```
            if L[i]<=p:
                L1.append(L[i]) # ajoute L[i] à L1
            else:
                L2.append(L[i]) # ajoute L[i] à L2
    L3=quicksort2(L1)+[p]+quicksort2(L2) # concaténation de 3 listes
    return L3
```

## 10.7

**a)**

```
def tri_comptage(L): # la fonction retourne L_tri la liste triée
                     # par ordre croissant de la liste L
    n=len(L)         # nombre d'éléments de L
    L_tri=[]
    p=L[0]           # recherche du maximum de L noté p
    for i in range(n):
        if L[i]>p:
            p=L[i]   # nouvelle valeur du maximum
    HISTO=[0 for i in range(p+1)]  # liste 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 la liste 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)
```

La liste `HISTO` vaut : [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 3, 0, 0, 1, 2, 0, 0, 0, 2].

**b)** 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 une liste 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.

**c)** On définit $T(n)$ le nombre d'opérations élémentaires :

- Ligne `n=len(L)` : deux opérations élémentaires (appel de `len(L)` et affectation).
- Ligne `L_tri=[]` : une opération élémentaire.
- Boucle `for i in range(n)` : à chaque étape, quatre 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=[0 for i in range(p+1)]` : $p+1$ opérations élémentaires.
- Boucle `for i in range(n)`. À chaque étape, quatre 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 deux 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.

**d)**

```
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 le graphique
```

**e)** On peut réduire le nombre d'éléments de la liste `HISTO` en calculant le minimum et le maximum de `L`, notés respectivement `mini` et `maxi`. La liste `HISTO` contient alors `maxi-mini+1` éléments au lieu de `p+1` éléments dans la question a).

```
def tri_comptage2(L):     # la fonction retourne L_tri la liste triée
                          # par ordre croissant de la liste L
    n=len(L)              # nombre d'éléments de L
```

```
    L_tri=[]
    mini, maxi=L[0], L[0]    # recherche du minimum et maximum de L
    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=[0 for i in range(maxi-mini+1)]
                 # liste contenant p+1 valeurs nulles
    # 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 la liste 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)
```

## 10.8

**a)**

```
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=[12, 35, 40, 14, 22, 13, 25, 16]
tri_bulles(L)  # appel de la fonction tri_bulles
print(L)       # on affiche la liste L car tri en place
```

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 : $i = n - 1 = 4$

On a une deuxième boucle `for j in range(1, i+1)` : $j$ varie entre 1 inclus et 5 exclu qui permet de comparer deux éléments consécutifs de `L` et de les échanger s'ils sont mal triés.

L'élément 8 va remonter comme une bulle jusqu'à l'indice 2 de la liste. L'élément 22 remonte comme une bulle jusqu'à l'indice $n - 1$ de la liste. On obtient : `L=[4, 2, 8, 6, 22]`. L'élément 22 est donc bien placé.

- Deuxième étape : $i = 3$. On considère uniquement les éléments : `4, 2, 8, 6` puisque la deuxième boucle fait varier *j* entre 1 inclus et 4 exclu (ou 3 inclus). On obtient : `L=[2, 4, 6, 8, 22]`.
- …
- Dernière étape : `i = 1`. On obtient : `L=[2, 4, 6, 8, 22]`.

> **Remarque**
>
> On peut parcourir la liste 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.

**b)** Il suffit de remplacer la ligne `if L[j-1]>L[j]` par `if L[j-1][0]>L[j][0]:`.

```
def tri_bulles2(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 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][0]>L[j][0]:
                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):

L=[[213, 'Brésil'], [1444, 'Chine'], [126, 'Japon'], \
   [111, 'Philippines'], [145, 'Russie']]
tri_bulles2(L)
print(L)
```

Le tri à bulles est comparatif et itératif.

La liste `P` est triée par ordre alphabétique. La liste triée par ordre croissant de la population est :

```
[[85, 'Iran'], [85, 'Turquie'], [111, 'Philippines'], [126, 'Japon'],
[145, 'Russie'], [213, 'Brésil'], [1444, 'Chine']]
```

Le tri à bulles est 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.

**c)** On définit $T(n)$ le nombre d'opérations élémentaires :

- Ligne `n=len(L)` : deux 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 sept 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]`, deux 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 $O\left(n^2\right)$.

**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.

**d)** 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`.

**Attention**

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.

# Graphes

CHAPITRE 11

Plan



## Thèmes abordés dans les exercices

- File, pile, deque, dictionnaire
- Matrice d'adjacence
- Liste d'adjacence
- Parcours en largeur (BFS, *Breadth First Search* en anglais)
- Parcours en profondeur (DFS, *Depth First Search* en anglais)

## Points essentiels du cours pour la résolution des exercices

- Utiliser une deque pour le parcours en largeur d'un graphe
- Utiliser une pile pour le parcours en profondeur d'un graphe
- Identifier les principes LIFO et FIFO
- Détecter si un graphe non orienté est connexe

# Les méthodes à retenir

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 $G$.

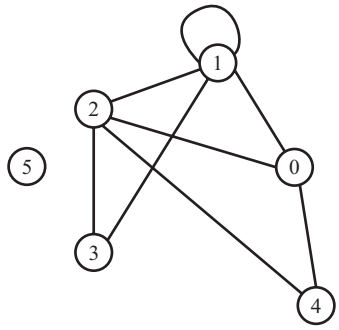
Si une arête relie les sommets $s$ et $s'$, on dit que **les sommets $s$ et $s'$ sont voisins ou adjacents**.

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.

**Simplification de vocabulaire**

Par la suite, on emploiera le terme de **chemin** au lieu de chaîne.

**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 **longueur 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 d'un 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 *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.

**Simplification de vocabulaire**

Par la suite, on emploiera le terme de **cycle** au lieu de circuit.

**Remarque**

Des arêtes reliant la même paire de sommets sont des arêtes multiples. Un graphe est simple s'il ne contient ni boucle ni arête multiple. 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

On utilise une liste de listes. Par exemple la matrice $\begin{pmatrix} 3 & 2 \\ 8 & 6 \end{pmatrix}$ est représentée par la liste `M` contenant deux listes de longueur 2 : `M=[[3, 2], [8, 6]]`. Chacune de ces listes de longueur 2 représente une ligne de la matrice.

On définit la variable `inf=1e10` qui représente une distance infinie.

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 cas :

1) Graphe non pondéré

- Graphe non orienté : si deux sommets différents *i* et *j* sont reliés par une arête, alors $M[i][j] = M[j][i] = 1$, sinon $M[i][j] = M[j][i] = 0$.
- Graphe orienté : s'il existe un arc allant de *i* vers *j*, alors $M[i][j] = 1$, sinon $M[i][j] = 0$.

2) Graphe pondéré (voir exercice « Algorithme de Dijkstra » dans le chapitre 12 « Recherche d'un plus court chemin »)

- Graphe non orienté : si deux sommets différents *i* et *j* sont reliés par une arête, alors $M[i][j] = M[j][i] =$ distance entre les sommets *i* et *j*, sinon $M[i][j] = M[j][i] = \inf$.
- Graphe orienté : s'il existe un arc allant de *i* vers *j*, alors $M[i][j]$ = distance entre les sommets d'origine *i* et d'extrémité *j*, sinon cette distance vaut l'infini : $M[i][j] = \inf$. Le poids des arcs peut être négatif (voir exercice « Algorithme de Bellman-Ford » dans le chapitre 12 « Recherche d'un plus court chemin »).

# Liste d'adjacence

On a plusieurs possibilités pour représenter une liste d'adjacence dans Python :

1) Dictionnaire

- Graphe non orienté : la clé associée à chaque sommet représente la liste des sommets adjacents.
- Graphe orienté : la clé associée à chaque sommet représente la liste des successeurs.

2) Liste

- Graphe non orienté : chaque élément de la liste contient un sommet et la liste des sommets adjacents.
- Graphe orienté : chaque élément de la liste contient un sommet et la liste des successeurs.

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 :

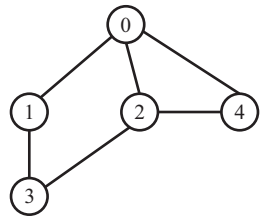
- Site web : chaque page est un sommet du graphe. Un lien hypertexte est une arête entre deux sommets.
- Réseau ferroviaire : chaque gare est un sommet. Les voies entre deux gares sont des arêtes.
- Réseau routier : chaque ville est un sommet. Les routes entre deux villes sont des arêtes.
- Réseau social : 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.

# 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 »).

### Parcours en largeur d'un graphe (BFS, *Breadth First Search* en anglais) – Principe FIFO

On considère un réseau social ayant 5 abonnés (0, 1, 2, 3, 4). Chaque abonné est représenté par un cercle et la relation « X est ami(e) avec Y » par une arête. On considère le graphe non orienté $G=(S,A)$ :




On définit la matrice d'adjacence $\left(M_{i,j}\right)_{0\le i,j\le 4}$ (appelée également matrice de distance) du graphe *G*, par :

Si deux sommets différents *i* et *j* sont reliés par une arête, alors $M[i][j]=M[j][i]=1$ sinon $M[i][j]=M[j][i]=0$.

La matrice d'adjacence est : $M = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{pmatrix}$.

Tous les éléments de la diagonale sont nuls d'après la définition de la matrice d'adjacence : $M[i][i] = 0$.

Les éléments sont symétriques par rapport à la diagonale puisque le graphe n'est pas orienté : $M[i][j] = M[j][i]$. 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.

On considère une deque `D` et une liste `VISITED` qui contient la liste des sommets explorés.

**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` **tous les sommets** *j* non explorés adjacents au sommet *i*.

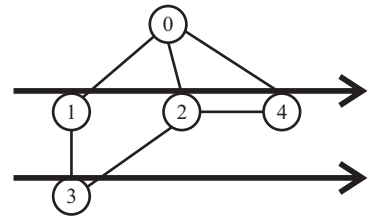
L'algorithme de parcours en largeur (ou BFS, *Breadth First Search* en anglais) permet de traiter les sommets adjacents à un sommet donné pour ensuite les explorer un par un. Il utilise une deque `D` dans laquelle il place le premier sommet à l'extrémité droite de `D` initialement vide et les sommets adjacents non explorés à l'extrémité droite de `D`. On utilise le principe FIFO (*First In*, *First Out* : « premier entré, premier sorti »).

```
def parcourslargeur(M, début):
    # la fonction permet un parcours en largeur de la matrice d'adjacence M
    # en partant de début et retournant VISITED la liste des sommets explorés
    from collections import deque  # module permettant d'utiliser les deques
    D=deque()          # deque vide
    D.append(début)    # début = 1er élément de la deque D
    VISITED=[début]    # début = 1er élément de la liste des sommets explorés
    n=len(M)           # nombre de sommets
    while len(D)!=0:
        i=D.popleft() # supprime le sommet i à l'extrémité gauche de D
        for j in range(n):
            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
    return VISITED

M=[[0,1,1,0,1],[1,0,0,1,0],[1,0,0,1,1],[0,1,1,0,0],[1,0,1,0,0]]
VISITED=parcourslargeur(M, 0)
print("Parcours en largeur d'un graphe")
print('Sommets explorés :', VISITED)
```

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`, on explore les sommets suivants : `VISITED = [0, 1, 2, 4, 3]`. On a bien réalisé un parcours en largeur.




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.

### Parcours en profondeur d'un graphe (DFS, *Depth First Search* en anglais) – Principe LIFO

On reprend le graphe étudié précédemment. On définit une pile `P` et une liste `VISITED` qui contient la liste des sommets explorés. 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 `VISITED` contient le sommet de départ : `VISITED =[début]`.

**Boucle tant que la pile `P` n'est pas vide** :

- **S'il existe un sommet** *j* non exploré adjacent au sommet *i* (situé en haut de la pile `P`) qui n'est pas dans la liste `VISITED`, alors on empile *j* dans `P` et on ajoute *j* dans la liste `VISITED`.

> **Attention**
>
> Lorsqu'on utilise des piles, il faut dépiler et empiler pour obtenir l'élément en haut de la pile.

- Si le sommet *i* (situé en haut de la pile `P`) ne possède pas de voisin non exploré, alors on dépile cet élément de la pile.

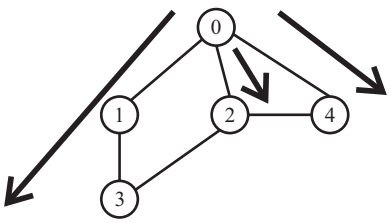
On utilise un flag `trouve` (de valeur `True` ou `False`) qui permet de savoir si le sommet *i* possède un voisin non exploré.

```
def parcoursprofondeur(M, début):
    # la fonction permet un parcours en profondeur de la matrice d'adjacence M
    # en partant de début et retournant VISITED la liste des sommets explorés
```

```
    P=[début]        # début = 1er élément de la pile P
    VISITED=[début]  # début = 1er élément de la liste des sommets explorés
    n=len(M)         # nombre de sommets
    while len(P)!=0: # tant que la pile P n'est pas vide
        i=P.pop()    # dépile pour récupérer l'élément i en haut de la pile
        P.append(i)  # empile i car il ne faut pas dépiler i à ce stade
        trouve=False
        j=0
        while j<n and trouve==False:
            if M[i][j]==1 and (j not in VISITED):
                # on cherche un voisin non exploré
                trouve=True       # on a trouvé un voisin non exploré
                P.append(j)       # ajoute ce sommet en haut de la pile
                VISITED.append(j) # marque ce voisin exploré
            j+=1
        if trouve==False:
            P.pop()  # dépile le haut de la pile
    return VISITED

M=[[0,1,1,0,1],[1,0,0,1,0],[1,0,0,1,1],[0,1,1,0,0],[1,0,1,0,0]]
VISITED=parcoursprofondeur(M, 0)
print("Parcours en profondeur d'un graphe")
print('Sommets explorés :', VISITED)
```

On utilise le principe LIFO (*Last In*, *First Out* : « dernier entré, premier sorti »).

Avec la fonction `parcoursprofondeur`, on explore les sommets suivants : `VISITED = [0, 1, 3, 2, 4]`. On a bien réalisé un parcours en profondeur.




Parcours en profondeur : 0 - 1 - 3 puis 2 puis 4

**Remarque**

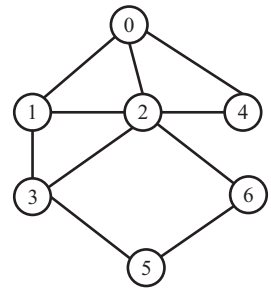
On empile uniquement un voisin non exploré dans le parcours en profondeur alors qu'on ajoute tous les voisins non explorés dans le parcours en largeur.

# Énoncés des exercices

## 11.1

**Réseau social, parcours en largeur, dictionnaire**

On considère un réseau social ayant 7 abonnés (0, 1, 2, 3, 4, 5, 6). Chaque abonné est représenté par un cercle et la relation « X est ami(e) avec Y » par une arête. On considère le graphe non orienté $G=(S,A)$ :




**a)** 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*.

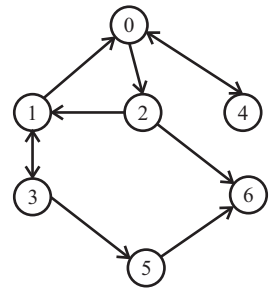
**b)** Écrire une fonction `parcourslargeur_dico` qui admet comme arguments un dictionnaire `dico` et un sommet de départ `début` permettant de parcourir en largeur le graphe. On utilise une deque pour parcourir en largeur le graphe. La fonction retourne la liste des sommets explorés.

**c)** Calculer la complexité de cet algorithme dans le pire des cas.

## 11.2

**Site web, parcours en largeur, 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 7 pages numérotées de 0 à 6. On considère le graphe orienté $G=(S,A)$ représentant la structure du site web.



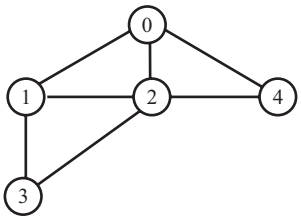

**a)** Construire la matrice d'adjacence *M* du graphe *G*.

**b)** Écrire une fonction `parcourslargeur_file` qui admet comme arguments une matrice d'adjacence `M` et un sommet de départ `début` permettant de parcourir en largeur un arbre. La fonction retourne la liste des sommets explorés.

## 11.3

**Test de connexité d'un graphe, parcours en largeur, arbre couvrant**

On considère un réseau social ayant 5 abonnés (0, 1, 2, 3, 4, 5, 6). Chaque abonné est représenté par un cercle et la relation « X est ami(e) avec Y » par une arête. On considère le graphe non orienté $G=(S,A)$ :




**a)** Construire la matrice d'adjacence *M* du graphe *G*.

**b)** Écrire une fonction `testgrapheconnexe_largeur` qui admet comme argument une matrice d'adjacence `M` et retourne `True` si le graphe est connexe et `False` sinon. On utilise une deque pour parcourir en largeur le graphe.

**c)** 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 n'utilise pas `PERE[0]` par la suite.
- La deque `D` contient le sommet de départ.
- La liste `VISITED` contient le sommet de départ.

**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.

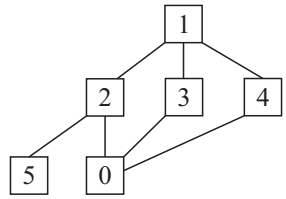
Écrire une fonction `arbrecouvrant_largeur` qui admet comme arguments une matrice d'adjacence `M` et un sommet de départ `début` permettant de récupérer une matrice d'adjacence représentant l'arbre couvrant correspondant au parcours en largeur pour un graphe non orienté et connexe.

**d)** Représenter l'arbre couvrant du graphe *G*.

## 11.4

**Recherche d'un cycle, graphe non orienté, 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.

**a)** Construire la matrice d'adjacence $M_{i,j}$ du graphe *G*. Si deux sommets différents *i* et *j* sont reliés par une arête, alors $M[i][j]=1$, sinon $M[i][j]=0$.

**b)** 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.

**c)** Écrire le programme principal permettant d'afficher si un graphe connexe et non orienté possède au moins un cycle. On pourra appeler la fonction `cycle_som` à un sommet quelconque du graphe.

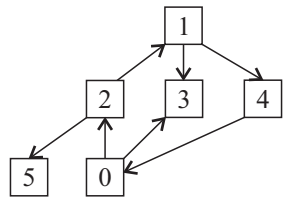
**d)** Écrire le programme principal permettant d'afficher si un graphe non orienté possède au moins un cycle. On pourra appeler la fonction `cycle_som` à chaque sommet du graphe.

## 11.5

■■■□

**Recherche d'un cycle, graphe orienté, 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 6 pages numérotées de 0 à 5. On considère le graphe orienté $G=(S,A)$ représentant la structure du site web.




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 gris. Après avoir traité en largeur tous les successeurs (qui deviennent gris) de ce sommet *i*, le sommet *i* est noir.

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.

**a)** Construire la matrice d'adjacence *M* du graphe *G*.

**b)** Écrire une fonction `cycle_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 cycle passant par le sommet *s*.

**c)** Écrire le programme principal permettant d'afficher si le graphe orienté possède un cycle passant par le sommet `début`.

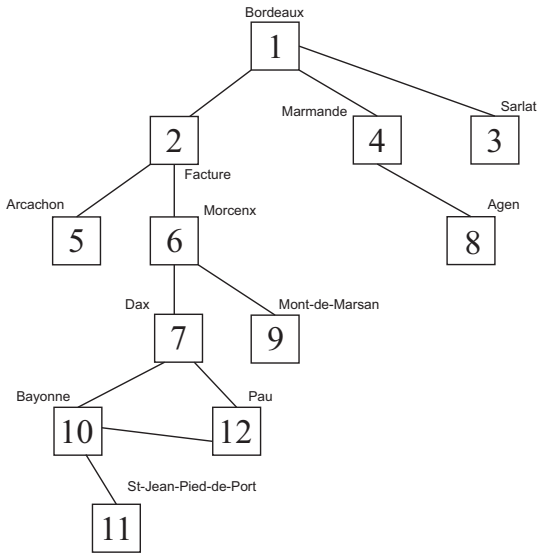
**d)** Écrire le programme principal permettant d'afficher si le graphe orienté possède au moins un cycle. On pourra appeler la fonction `cycle_sommet` à chaque sommet du graphe.

## 11.6

■■■□

**Algorithme récursif du parcours en largeur**

On considère le graphe non orienté $G = (S, A)$ représentant la structure du réseau ferroviaire de Bordeaux. Chaque gare est un sommet. Les voies entre deux gares sont des arêtes.




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`.

**a)** 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.

**b)** Écrire une fonction récursive `BFS_rec` qui admet comme arguments un dictionnaire `dico`, une deque `D` et une liste `PARCOURS`.

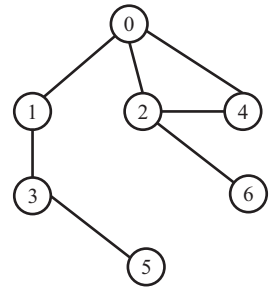
**c)** Le sommet de départ vaut 1. Que vaut `PARCOURS` après l'appel de la fonction `BFS_rec (dico, D, PARCOURS)` ?

## 11.7

■■□□

**Réseau social, parcours en profondeur, dictionnaire**

On considère un réseau social ayant 7 abonnés (0, 1, 2, 3, 4, 5, 6). Chaque abonné est représenté par un cercle et la relation « X est ami(e) avec Y » par une arête. On considère le graphe non orienté $G=(S,A)$ :




**a)** 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*.

**b)** Écrire une fonction itérative `parcoursprofondeur_dico` qui admet comme arguments un dictionnaire `dico` et un sommet de départ `début` permettant de parcourir en profondeur le graphe. La fonction retourne la liste des sommets explorés.

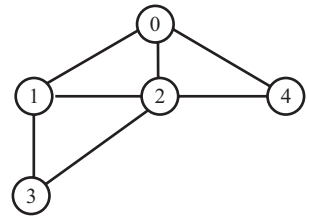
**c)** Que retourne `parcoursprofondeur_dico(dico, 0)` ?

## 11.8

■■■■

**Test de connexité d'un graphe, parcours en profondeur, arbre couvrant**

On considère un réseau social ayant 5 abonnés (0, 1, 2, 3, 4). Chaque abonné est représenté par un cercle et la relation « X est ami(e) avec Y » par une arête. On considère le graphe non orienté $G=(S,A)$ :




**a)** Construire la matrice d'adjacence *M* du graphe *G*.

**b)** Écrire une fonction `testgrapheconnexe_prof` 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.

**c)** 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`, `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** :

- S'il existe un sommet *i* non exploré adjacent au sommet *x* (situé en haut de la pile `P`) qui n'est pas dans la liste `PARCOURS`, alors on empile *i* dans `P`, on ajoute *i* dans la liste `PARCOURS` et on ajoute *x* dans la liste `PERE`.
- Si le sommet *x* (situé en haut de la pile `P`) ne possède pas de voisin non exploré, alors on dépile cet élément de la pile.

Écrire une fonction `arbrecouvrant_prof` 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.

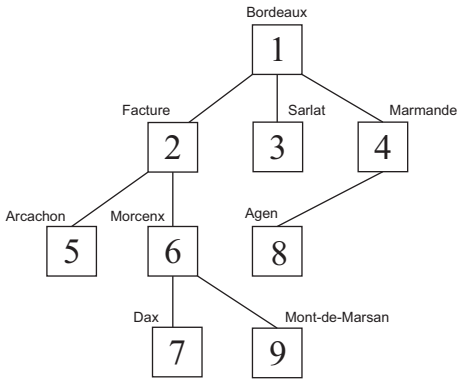
**d)** Représenter l'arbre couvrant du graphe *G*.

## 11.9

■■□□

**Algorithme récursif du parcours en profondeur**

On considère le graphe non orienté $G = (S, A)$ représentant la structure du réseau ferroviaire de Bordeaux. Chaque gare est un sommet. Les voies entre deux gares sont des arêtes.




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`.

**a)** 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.

**b)** Écrire une fonction récursive `profondeur_rec` qui admet comme arguments un dictionnaire `dico`, un sommet de départ `s` et une liste `PARCOURS`.

**c)** Qu'affiche le programme suivant ?

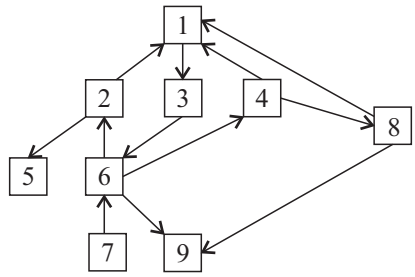
```
profondeur_rec(dico, 1, [])
print(PARCOURS)
```

Représenter l'arbre des appels de la fonction récursive `profondeur_rec(dico, 1)`.

## 11.10

**Recherche d'un cycle, parcours en profondeur récursif**

On considère le graphe orienté $G=(S,A)$ :




On utilise le dictionnaire `couleur` pour mémoriser la couleur des sommets. Un sommet est blanc lorsqu'il n'a pas été traité. Le dictionnaire `couleur` contient initialement tous les sommets blancs.

Lorsqu'on commence à traiter un sommet *i*, il est gris. Après avoir traité en profondeur tous les successeurs de ce sommet *i*, le sommet *i* est noir.

La variable globale `trouve` est initialisée à `False`. La variable globale `début` représente le sommet de départ.

**a)** Définir un dictionnaire `dico` représentant le graphe *G*. La clé associée à chaque sommet représente la liste des successeurs.

**b)** Écrire une fonction récursive `cycle_rec_sommet` qui admet comme arguments un dictionnaire `dico`, un sommet de départ `s` et le dictionnaire `couleur`. Cette fonction parcourt en profondeur le graphe *G*.

La variable globale `trouve` vaut `True` lorsqu'un des successeurs d'un sommet est grisé et que ce successeur est le sommet de départ `début`.

**c)** Écrire le programme principal permettant d'afficher si le graphe orienté possède un cycle passant par le sommet `début`.

**d)** Écrire le programme principal permettant d'afficher si le graphe orienté possède au moins un cycle. On pourra appeler la fonction `cycle_rec_sommet` à chaque sommet du graphe.

# Du mal à démarrer ?

## 11.1

**a)** Le dictionnaire contient 7 clés. La valeur de chaque clé contient tous les sommets adjacents.

**b)** Créer une deque `D` contenant uniquement `début` et une liste `VISITED` contenant uniquement `début`. Écrire une boucle tant que la deque `D` n'est pas vide.

**c)** Compter le nombre d'opérations élémentaires.

## 11.2

**a)** S'il existe un arc allant de *i* vers *j*, alors $M[i][j]=1$, sinon $M[i][j]=0$.

**b)** Créer une file `F` contenant `début` et une liste `VISITED` contenant uniquement `début`. Écrire une boucle tant que la file `F` n'est pas vide.

## 11.3

**a)** Si deux sommets différents *i* et *j* sont reliés par une arête, alors $M[i][j]=M[j][i]=1$, sinon $M[i][j]=M[j][i]=0$.

**b)** Créer une deque `D` contenant uniquement `début` et une liste `VISITED` contenant uniquement `début`. Écrire une boucle tant que la deque `D` n'est pas vide.

Le graphe est connexe si le nombre d'éléments de la liste `PARCOURS` est égal au nombre de sommets du graphe *G*.

**c)** Un graphe non orienté, connexe et acyclique est un arbre. 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*.

Écrire une boucle `for k in range` commençant à 1. `VISITED[k]` et `PERE[k]` permettent de remplir deux éléments de la matrice d'adjacence de l'arbre couvrant.

## 11.4

**a)** Si deux sommets différents *i* et *j* sont reliés par une arête, alors $M[i][j]=M[j][i]=1$, sinon $M[i][j]=M[j][i]=0$.

**b)** Créer une deque `D` contenant uniquement `début` et une liste `couleur` dont tous les éléments sont blancs. Écrire une boucle tant que la deque `D` n'est pas vide.

**c)** Un cycle est un chemin simple tel 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. Appeler la fonction `cycle_com` avec le sommet 0 par exemple.

**d)** Appeler la fonction `cycle_com` à chaque sommet du graphe.

## 11.5

**a)** S'il existe un arc allant de *i* vers *j*, alors $M[i][j]=1$, sinon $M[i][j]=0$.

**b)** Créer une deque `D` contenant uniquement `début` et une liste `couleur` dont tous les éléments sont blancs. Écrire une boucle tant que la deque `D` n'est pas vide.

**c)** Appeler la fonction `cycle_sommet` au sommet début.

**d)** Appeler la fonction `cycle_sommet` à chaque sommet du graphe.

## 11.6

**a)** Le dictionnaire contient 12 clés. La valeur de chaque clé contient tous les sommets adjacents.

**b)** `len(D)==0` est une condition d'arrêt de la fonction récursive.

## 11.7

**a)** Le dictionnaire contient 7 clés. La valeur de chaque clé contient tous les sommets adjacents.

**b)** Créer une pile `P` contenant l'élément `début` et une liste `PARCOURS` contenant l'élément `début`. Définir un flag `trouve` initialisé à `False`. On empile uniquement un voisin non exploré dans le parcours en profondeur.

### 11.8

**a)** Si deux sommets différents $i$ et $j$ sont reliés par une arête, alors $M[i][j] = M[j][i] = 1$, sinon $M[i][j] = M[j][i] = 0$.

**b)** Créer une pile `P` contenant l'élément `début` et une liste `PARCOURS` contenant l'élément `début`. Définir un flag `trouve` initialisé à `False`. On empile uniquement un voisin non exploré dans le parcours en profondeur.

Le graphe est connexe si le nombre d'éléments de la liste `PARCOURS` est égal au nombre de sommets du graphe *G*.

**c)** Un graphe non orienté, connexe et acyclique est un arbre. 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*.

### 11.9

**a)** Le dictionnaire contient 9 clés. La valeur de chaque clé contient tous les sommets adjacents.

**b)** Définir une liste contenant tous les sommets adjacents au sommet `s`. Tester si chaque élément de cette liste est dans `PARCOURS`.

### 11.10

**a)** Le dictionnaire contient 9 clés. La valeur de chaque clé contient la liste des successeurs.

**b)** Définir une liste contenant tous les sommets adjacents au sommet `s`. Tester la couleur de chaque élément de cette liste. Si la couleur est blanche, appeler récursivement la fonction `cycle_rec_sommet`.

**c)** On part du sommet `début` et on explore en profondeur le graphe. Lors de l'exploration en profondeur, si on trouve un sommet qui a pour successeur `début`, alors on a trouvé un cycle.

**d)** Pour savoir s'il existe au moins un cycle dans un graphe, il suffit de vérifier pour chaque sommet du graphe s'il existe un cycle passant par ce sommet.

# Corrigés des exercices

### 11.1

**a)**

```
dico={0:[1, 2, 4], 1:[0, 2, 3], 2:[0, 1, 3, 4, 6], 3:[1, 2, 5], 4:[0, 2],\
   5:[3, 6], 6:[2, 5]}
```

Le sommet 2 a cinq sommets adjacents 0, 1, 3, 4 et 6. La valeur de la clé 2 est la liste des sommets adjacents : [0, 1, 3, 4, 6].

**b)** 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 *elt* non explorés adjacents au sommet *i*.

```
def parcourslargeur_dict(dico, début):
    # la fonction permet 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]                 # liste des sommets explorés
    while len(D)!=0:
        i=D.popleft()               # supprime le sommet i à l'extrémité gauche de D
        L=dico[i]
        for elt in L:
            if elt not in VISITED:
                D.append(elt)       # ajoute elt à l'extrémité droite de D
                VISITED.append(elt) # le sommet elt a été exploré
    return VISITED
```

On utilise le principe FIFO (*First In*, *First Out* : « premier entré, premier sorti »). La liste `VISITED` contient tous les sommets explorés.

**c)** 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 elt in L` est exécutée au maximum $n$ fois.

La complexité dans le pire des cas pour un graphe représenté par un dictionnaire est quadratique en $O\left(n^2\right)$.

## 11.2

**a)** La matrice d'adjacence est : $M = \begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$.

```
M=[[0,0,1,0,1,0,0], [1,0,0,1,0,0,0],[0,1,0,0,0,0,1],  \
  [0,1,0,0,0,1,0],[1,0,0,0,0,0,0],[0,0,0,0,0,0,1],[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 : $M[i,i] = 0$.

S'il existe un arc allant de $i$ vers $j$, alors $M[i,j] = 1$, sinon $M[i,i] = 0$.

On peut déduire de la quatrième ligne de la matrice que le sommet $i = 3$ est relié aux sommets : 1 et 5.

**b)** Les étapes de l'algorithme de parcours en largeur sont les suivantes :

**Initialisation de l'algorithme** :

- Mettre le sommet de départ dans la file `F` initialement vide.
- La liste `VISITED` contient le sommet de départ : `VISITED=[début]`.

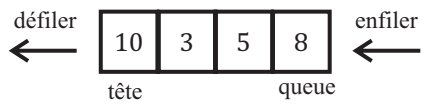
**Boucle tant que la file `F` n'est pas vide** :

- Défiler un élément de la file `F`.
- Ajouter dans `VISITED` et enfiler dans `F` les sommets non explorés adjacents au sommet défilé précédemment.

```
def parcourslargeur_file(M, début):
    # la fonction permet un parcours en largeur de la
    # matrice d'adjacence en partant de début et retournant
    # VISITED la liste des sommets explorés
    n=len(M)          # nombre de sommets
    F=[]              # file vide (modélisée par une liste dans Python)
    F.append(début)   # ajoute le sommet de départ
    VISITED=[début]   # liste des sommets explorés
    while F!=[]:
        i=F.pop(0)
        for j in range(n):
            if (M[i][j]!=0) and (j not in VISITED):
                F.append(j)        # ajoute j à la file F
                VISITED.append(j)  # le sommet j a été exploré
    return VISITED
```

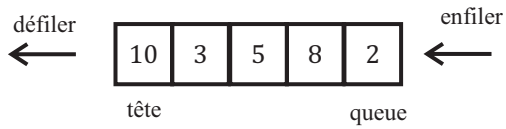
**Remarque**

On utilise une liste `F` pour modéliser une file. Par exemple `F = [10, 3, 5, 8]`.
Si on ajoute l'élément 2 (ou si on enfile l'élément 2), on obtient `F = [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.




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`.

Ne pas confondre avec `F.pop()`, qui permet de supprimer le dernier élément de la liste `F`.

## 11.3

**a)** La matrice d'adjacence est : $M = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{pmatrix}$.

```
M=[[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 : $M[i,i] = 0$.

Les éléments sont symétriques par rapport à la diagonale puisque le graphe n'est pas orienté : $M[i][j] = M[j][i]$. 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.

**b)** 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`.

Un graphe non orienté 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 un chemin. Il n'y a pas de sommet isolé. Pour savoir si le graphe *G* est connexe, on récupère la liste des sommets explorés avec l'algorithme de parcours en largeur. 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
    n=len(M)                       # nombre de sommets
    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)==n:
        return True
    else:
        return False
```

La liste `VISITED` de la fonction `testgrapheconnexe_largeur` donne la liste des tous les sommets explorés avec l'algorithme BFS.

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 11.8 « Test de connexité d'un graphe, parcours en profondeur, arbre couvrant ».

**c)** On utilise le parcours en largeur pour construire un arbre inclus dans le graphe *G* et qui relie tous les sommets de ce graphe. 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=[[0 for j in range(n)] for i in range(n)]
    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**

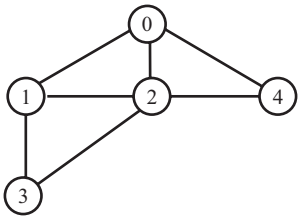
On peut également utiliser l'algorithme du parcours en profondeur pour construire un arbre couvrant. Voir exercice 11.8 « Test de connexité d'un graphe, parcours en profondeur, arbre couvrant ».

**d)** La liste `VISITED` vaut : [0, 1, 2, 4, 3].

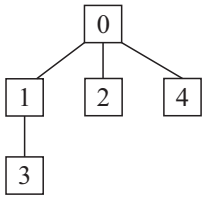
La liste `PERE` vaut : [None, 0, 0, 0, 1].

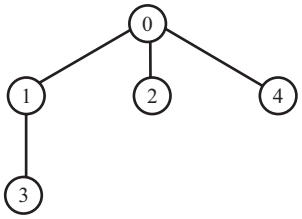
On obtient la matrice d'adjacence représentant l'arbre couvrant : $\begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}$.

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.

## 11.4

**a)** La matrice d'adjacence est : $M = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$.

```
M=[[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]]
```

**Remarque**

Tous les éléments de la diagonale sont nuls d'après la définition de la matrice d'adjacence : $M[i,i] = 0$.

Les éléments sont symétriques par rapport à la diagonale puisque le graphe n'est pas orienté : $M[i][j] = M[j][i]$.

**b)** Il y a des différences dans la recherche de cycle pour les graphes orientés et les graphes non orientés :

- Pour un graphe orienté (voir exercice 11.5 « Recherche d'un cycle, graphe orienté, 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 cycle. La condition pour trouver un cycle 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 cycle 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:`.

  Un chemin 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=len(M)                      # 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):
            # print(PERE,x,i)
            if M[x][i]>0 and couleur[i]!="blanc" and PERE[x]!=i:
                                # on a trouvé un chemin 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
```

**c)** 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)
```

**d)** 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=len(M)             # 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 est 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.')
```

## 11.5

**a)** La matrice d'adjacence est : $M = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$.

```
M=[[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**

S'il existe un arc allant de $i$ vers $j$, alors $M[i,j] = 1$ sinon $M[i,i] = 0$.
Tous les éléments de la diagonale sont nuls d'après la définition de la matrice d'adjacence : $M[i,i] = 0$.
Comme le graphe est orienté, la matrice n'est pas nécessairement symétrique.

**b)** Un chemin est simple si tous les arcs du chemin sont différents. Un cycle est un chemin simple tel que le sommet d'arrivée est le même que le sommet de départ.

Pour savoir s'il existe un cycle dans un graphe, il suffit de vérifier pour chaque sommet du graphe s'il existe un cycle passant par ce sommet.

```
def cycle_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=len(M)                # nombre de sommets du graphe
    couleur=["blanc" for i in range(n)]   # les sommets non traités sont blancs
    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 cycle
    return False                         # on n'a pas trouvé de cycle
```

**Remarque**

On pourrait supprimer la ligne `couleur[x]="noir"` puisqu'on ne fait pas la différence entre le noir et le gris.

**c)**

```
début=0
if cycle_sommet (M, début)==True:
    print('Le graphe orienté possède un cycle passant par le sommet', début)
else:
    print('Le graphe orienté ne possède pas de cycle passant par le sommet', début)
```

Le programme Python affiche : « Le graphe orienté possède un cycle passant par le sommet 0 ».

**d)** On appelle la fonction `cycle_sommet` à chaque sommet du graphe. Si on trouve un cycle passant par un sommet, alors le graphe possède au moins un cycle.

```
def rec_cycle_largeur(M): # la fonction retourne False si le
    # graphe non orienté (pour la matrice M)
    # ne possède pas de cycle
    n=len(M)              # nombre de sommets du graphe
    for i in range(n):    # i varie entre 0 inclus et n exclu
        if cycle_sommet (M, i)==True:
            return True   # quitte la fonction si un cycle est trouvé
    return False
if rec_cycle_largeur(M)==True:
    print('Le graphe orienté possède au moins un cycle.')
else:
    print('Le graphe non orienté ne possède pas de cycle.')
```

## 11.6

**a)**

```
dico={1:[2, 3, 4], 2:[1, 5, 6], 3:[1], 4:[1, 8], 5:[2],     \
   6:[2, 7, 9], 7:[6, 10, 12], 8:[4], 9:[6],10:[7, 11, 12], \
   11:[10], 12:[7, 10]}
```

Le sommet 7 est relié aux sommets 6, 10 et 12. La valeur de la clé 7 est la liste des sommets adjacents [6, 10, 12].

**b)** 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 »).

- 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(PARCOURS)
```

**c)** On obtient : `PARCOURS = [1, 2, 3, 4, 5, 6, 8, 7, 9, 10, 12, 11].`

## 11.7

**a)**

```
dico={0:[1, 2, 4], 1:[0, 3], 2:[0, 4, 6], 3:[1, 5], 4:[0, 2],\
   5:[3], 6:[2]}
```

Le sommet 2 a trois sommets adjacents 0, 4 et 6. La valeur de la clé 2 est la liste des sommets adjacents : [0, 4, 6].

**b)** On considère une pile `P` et une liste `PARCOURS` qui contient la liste des sommets explorés. 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 :**

- S'il existe un sommet *elt* non exploré adjacent au sommet *x* (situé en haut de la pile `P`) qui n'est pas dans la liste `PARCOURS`, alors on empile *elt* dans `P` et on ajoute *elt* dans la liste `PARCOURS`.
- Si le sommet *x* (situé en haut de la pile `P`) ne possède pas de voisin non exploré, alors on dépile cet élément de la pile.

On utilise un flag `trouve` (de valeur `True` ou `False`) qui permet de savoir si le sommet *x* possède un voisin non exploré.

```
def parcoursprofondeur_dico(dico, début):
    # la fonction permet 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 en haut 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
        i=0
        while i<len(L) and trouve==False:
            if L[i] not in PARCOURS:  # on cherche un voisin non exploré
                trouve=True           # on a trouvé un voisin non exploré
                                      # le sommet elt vaut L[i]
                P.append(L[i])        # on ajoute ce sommet en haut de la pile
                PARCOURS.append(L[i]) # on 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(dico, 0)
print(PARCOURS)
```

**Remarque**

On utilise très souvent trois opérations de base avec les piles : « empiler », « dépiler » et « teste si la pile est vide ». Les lignes suivantes permettent de récupérer le sommet *x* en haut 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]     # L = liste de sommets adjacents à x
```

**c)** 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 »).

On explore les sommets suivants : [0, 1, 3, 5, 2, 4, 6].

Le sommet 2 a deux sommets adjacents. On commence par explorer en profondeur 4 puis 6 puisque les sommets sont triés par ordre croissant dans les valeurs des clés de `dico`.

**Remarque**

Dans l'algorithme du parcours en profondeur, on peut choisir d'autres parcours d'exploration :

- Le sommet 0 a trois sommets adjacents.
- On a commencé par explorer en profondeur 1, 3 et 5 mais on aurait pu explorer un autre parcours, par exemple 2 et 6.

## 11.8

**a)** La matrice d'adjacence est : $M = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{pmatrix}$.

```
M=[[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 : $M[i,i] = 0$.

Les éléments sont symétriques par rapport à la diagonale puisque le graphe n'est pas orienté : $M[i][j] = M[j][i]$. 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.

**b)** 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_prof(M):
    # la fonction retourne True si le graphe est connexe et False sinon
    # pour la matrice d'adjacence M
    n=len(M)          # nombre de sommets
    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 en haut de la pile
        P.append(x)  # empile x car il ne faut pas dépiler x à ce stade
        trouve=False
        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()      # 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
```

La liste `PARCOURS` de la fonction `testgrapheconnexe_prof` donne la liste des tous les sommets explorés avec l'algorithme en profondeur.

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.

**c)** 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_prof(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=len(M)          # nombre de sommets
    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 en haut de la pile
        P.append(x)   # empile x car il ne faut pas dépiler x à ce stade
        trouve=False
        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 x 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
    print('Parcours :',PARCOURS)
    print('Pere :',PERE)

    # Construction de l'arbre couvrant
    mat=[[0 for j in range(n)] for i in range(n)]
    for k in range(1, len(PARCOURS)): # k varie entre 1 inclus
        # et len(PARCOURS) exclu.
        # PARCOURS[0] n'a pas de père. C'est le premier élément du parcours
        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**

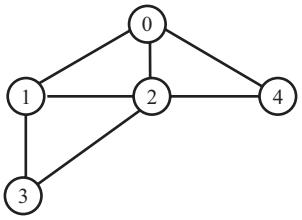
On peut également utiliser l'algorithme du parcours en largeur pour explorer tous les sommets du graphe. Voir exercice 11.3 « Test de connexité d'un graphe, parcours en largeur, arbre couvrant ».

**d)** La liste `PARCOURS` vaut : [0, 1, 2, 3, 4].

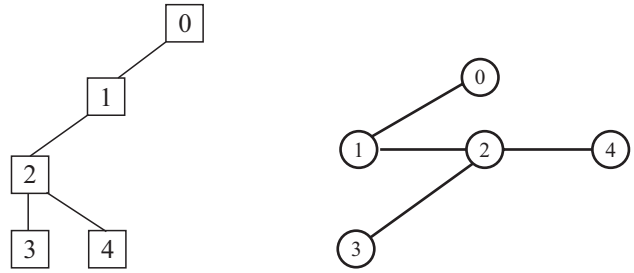
La liste `PERE` vaut : [-1, 0, 1, 2, 2].

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.

## 11.9

**a)**

```
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 (gare de Facture) est relié aux sommets 1 (Bordeaux), 5 (Arcachon) et 6 (Morcenx). La valeur de la clé 2 est la liste des sommets adjacents [1, 5, 6].

**b)** 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 définit une liste `L` contenant la liste des sommets adjacents à `s`. La boucle `for elt in L` permet de parcourir tous les éléments de `L`. Dès qu'on rencontre un sommet non exploré, on appelle la fonction récursive. On va ainsi le plus profond possible pour chaque branche.

Ce n'est pas nécessaire de faire un test pour la condition d'arrêt. Dès qu'on a parcouru entièrement la liste `L`, on est à la condition d'arrêt de la fonction récursive.

```
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
```

**c)** On considère le programme principal suivant :

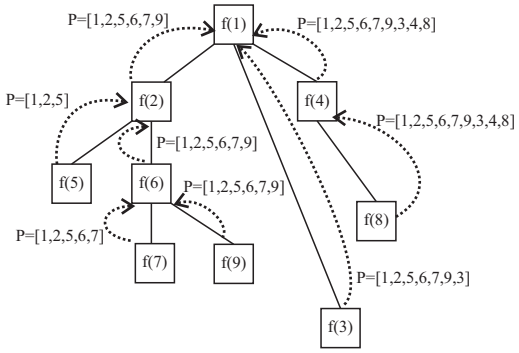
```
profondeur_rec(dico, 1, [])
print(PARCOURS)
```

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-dessous représente les différents appels de la fonction `profondeur_rec` que l'on note `f`.




## 11.10

**a)**

```
dico={1:[3], 2:[1, 5], 3:[6],
      4:[1, 8], 5:[], 6:[2, 4, 9],
      7:[6], 8:[1, 9], 9:[]}
```

Le sommet 4 a deux successeurs : 1 et 8. La valeur de la clé 4 est la liste des successeurs [1, 8]. Le sommet 6 n'est pas le successeur du sommet 4.

**b)**

```
def cycle_rec_sommet(dico, s, couleur):
    # la fonction permet un parcours en profondeur du dictionnaire dico
    # en partant de s
    # couleur[i]="blanc" pour un sommet i non traité
    global trouve, début           # variables globales
    couleur[s]="gris"              # sommet en cours de traitement
    L=dico[s]                      # liste des sommets adjacents
    for elt in L:
        if couleur[elt]=="blanc":  # sommet non traité
            cycle_rec_sommet(dico, elt, couleur)  # appel récursif
        elif elt==début:  # le successeur de s est grisé et on revient à début
            trouve=True   # on a trouvé un cycle
    couleur[s]="noir"     # on a exploré en profondeur ce sommet
```

**Remarque**

On pourrait supprimer la ligne `couleur[s]="noir"` puisqu'on ne fait pas la différence entre le noir et le gris.

**c)**

```
couleur={1:"blanc", 2:"blanc", 3:"blanc", 4:"blanc",\
   5:"blanc", 6:"blanc", 7:"blanc", 8:"blanc", 9:"blanc"}
    # les sommets non traités sont blancs
début=9
trouve=False
cycle_rec_sommet(dico, début , couleur)
if trouve==True:
    print('Le graphe orienté possède un cycle passant' \
          ' par le sommet', début )
else:
    print('Le graphe orienté ne possède pas de cycle' \
          ' passant par le sommet', début )
```

Le programme Python affiche : « Le graphe ne possède pas de cycle passant par le sommet 9 ».

Si on applique le programme précédent avec `début` = 6, le programme affiche : « Le graphe orienté possède un cycle passant par le sommet 6 ».

**Remarque**

On peut utiliser le parcours en largeur ou en profondeur pour la recherche d'un cycle.

**d)** Principe de l'algorithme :

- Appeler la fonction `cycle_rec_sommet` à chaque sommet du graphe.
- Si on trouve un cycle passant par un sommet, alors le graphe possède au moins un cycle.

```
couleur={1:"blanc", 2:"blanc", 3:"blanc", 4:"blanc",\
   5:"blanc", 6:"blanc", 7:"blanc", 8:"blanc", 9:"blanc"}
    # les sommets non traités sont blancs
trouve=False
i=1
while trouve==False and i<=len(dico):
    début=i    # on considère le i
    cycle_rec_sommet(dico, début, couleur)
        # on cherche si un cycle passe par le sommet début
    i+=1       # ou i=i+1
if trouve==True:
    print('Le graphe orienté possède au moins un cycle.')
else:
    print('Le graphe orienté ne possède pas de cycle.')
```

# Recherche d'un plus court chemin (sauf TSI et TPC)

CHAPITRE 12

Plan



## Thèmes abordés dans les exercices

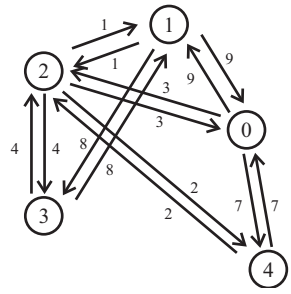
- Recherche d'un plus court chemin
- Algorithme de Dijkstra
- Algorithme A*
- Notion d'heuristique
- Fonction d'évaluation

## Points essentiels du cours pour la résolution des exercices

- Algorithme glouton
- Sous-graphe
- Utiliser une heuristique dont le coût évalué est inférieur au coût réel
- Définir une fonction d'évaluation
- Calculer une distance de Manhattan

# Les méthodes à retenir

On considère le graphe orienté $G=(S,A)$ pondéré avec des poids positifs 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 entre deux villes (ou sommets). On définit la matrice d'adjacence $\left(M_{i,j}\right)_{0\leq i,j\leq n-1}$ (appelée également **matrice de distance**) du graphe $G$, définie par : $\left(M_{i,j}\right)_{0\leq i,j\leq n-1}$ pour tous les indices $i$, $j$, $M_{i,j}$ 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=1e10` qui représente une distance infinie.




La matrice d'adjacence est : $M = \begin{pmatrix} 0 & 9 & 3 & \infty & 7 \\ 9 & 0 & 1 & 8 & \infty \\ 3 & 1 & 0 & 4 & 2 \\ \infty & 8 & 4 & 0 & \infty \\ 7 & \infty & 2 & \infty & 0 \end{pmatrix}$.

On cherche à déterminer un 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 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`.

# Algorithme de Dijkstra

### Étape d'initialisation

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`.

### Algorithme

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 de départ est la plus petite. Cette ville définit alors la nouvelle valeur de la variable `position` et cette ville devient sélectionnée.

```
inf=1e10      # variable représentant l'infini
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 un 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=len(M)                # nombre de villes = nombre 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]
```

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.

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.

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. L'algorithme de Dijkstra est donc un algorithme glouton.

# Recherche d'un plus court chemin

On applique l'algorithme de Dijkstra pour se rendre de la ville 3 à la ville 0 :

```
# initialisation du programme
M=[[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("Chemin suivi : ", L)   # liste des villes
print("Distance parcourue = ", dist)
```

Le programme Python affiche :

- Chemin suivi : [3, 2, 0]
- Distance parcourue = 7.0

### Étape d'initialisation

- Ville de départ = `départ` = 3 et ville d'arrivée = `arrivée` = 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 : $\text{VILLES} = \left[[-1, \text{inf}, \text{False}], [-1, \text{inf}, \text{False}], [-1, \text{inf}, \text{False}], [-1, 0, \text{True}], [-1, \text{inf}, \text{False}]\right]$. 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 un 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* = $\text{VILLES}[\text{position}][1]+\text{M}[\text{position}][\text{i}]$) et la distance depuis la ville de départ ($\text{VILLES}[i][1]$).
- 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 : $\text{VILLES}=\left[[-1,\text{inf},\text{False}],[3,8,\text{False}],[3,4,\text{False}],[-1,0,\text{True}],[-1,\text{inf},\text{False}]\right]$.
- 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 $\text{VILLES}=\left[[-1,\text{inf},\text{False}],[3,8,\text{False}],[3,4,\text{True}],[-1,0,\text{True}],[-1,\text{inf},\text{False}]\right]$.

**2e itération : `position` = 2**

- On parcourt les villes `X` non sélectionnées : 0, 1, 4.
  On obtient alors : $\text{VILLES}=\left[[2,7,\text{False}],[2,5,\text{False}],[3,4,\text{True}],[-1,0,\text{True}],[2,6,\text{False}]\right]$.
- 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 $\text{VILLES}=\left[[2,7,\text{False}],[2,5,\text{False}],[3,4,\text{True}],[-1,0,\text{True}],[2,6,\text{False}]\right]$.

**3e itération : `position` = 1**

- On parcourt les villes `X` non sélectionnées : 0, 4.
  On obtient alors : $\text{VILLES}=\left[[2,7,\text{False}],[2,5,\text{False}],[3,4,\text{True}],[-1,0,\text{True}],[2,6,\text{False}]\right]$.
- 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 $\text{VILLES}=\left[[2,7,\text{False}],[2,5,\text{False}],[3,4,\text{True}],[-1,0,\text{True}],[2,6,\text{False}]\right]$.

**4e itération : position = 4**

- On parcourt les villes `X` non sélectionnées : 0.
  On obtient alors : $\text{VILLES}=\left[[2,7,\text{False}],[2,5,\text{False}],[3,4,\text{True}],[-1,0,\text{True}],[2,6,\text{False}]\right]$.
- 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 0.
  On obtient alors : `position` = 0 et $\text{VILLES}=\left[[2,7,\text{False}],[2,5,\text{False}],[3,4,\text{True}],[-1,0,\text{True}],[2,6,\text{False}]\right]$.

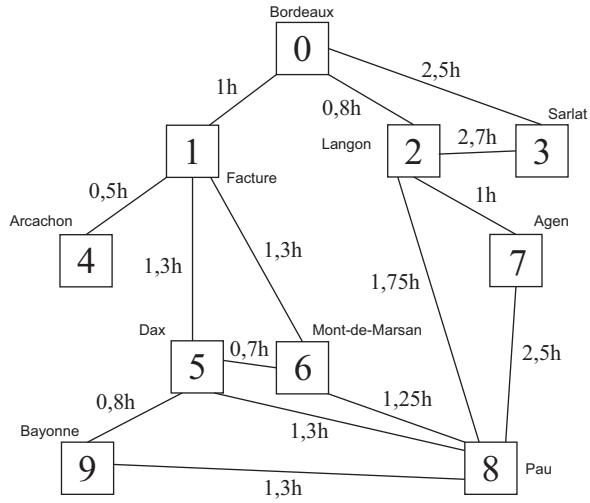
Pour obtenir le trajet, on part de la ville d'arrivée, la distance minimale entre la ville de départ et la ville d'arrivée est obtenue par $\text{VILLES}[\text{arrivée}][1]=7$. Pour obtenir le trajet, on obtient la ville précédente avec $\text{VILLES}[\text{arrivée}][0]=2$ et, de proche en proche, on remonte à la ville de départ.

# Énoncés des exercices

## 12.1

**Algorithme de Dijkstra (sauf TSI et TPC)**

On considère le graphe non orienté $G = (S, A)$, où le nombre situé sur l'arête joignant deux villes (ou sommets) est le temps de trajet en heures :




`L.reverse()` permet d'inverser les éléments de la liste `L`. On définit la variable `inf=1e10` qui représente un temps de trajet infini.

**a)** Construire la matrice d'adjacence $\left(M_{i,j}\right)_{0 \leq i,j \leq n-1}$ (appelée également matrice de distance) du graphe $G$, définie par : pour tous les indices $i$, $j$, $M_{i,j}$ représente le temps de trajet entre les sommets d'origine $i$ et d'extrémité $j$. Lorsque les sommets ne sont pas reliés, ce temps de trajet vaut l'infini.

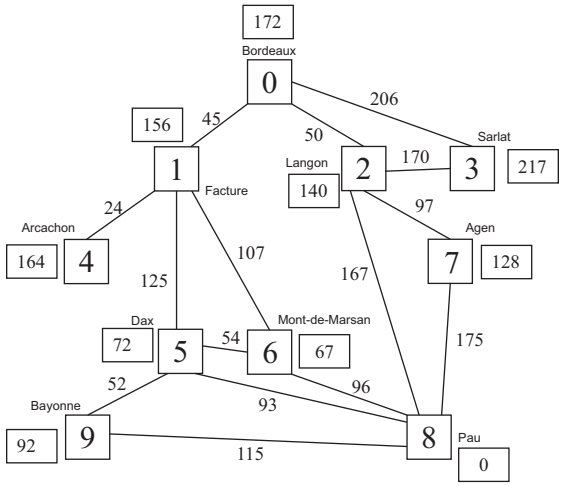
**b)** Écrire le programme Python permettant de déterminer un plus court chemin entre la ville de départ 0 et la ville d'arrivée 8 en utilisant l'algorithme de Dijkstra. Le programme affichera le chemin suivi ainsi que le temps de trajet en heures.

**c)** Calculer la complexité de cet algorithme dans le pire des cas.

## 12.2

**Algorithme A* (sauf TSI et TPC)**

On considère le graphe orienté $G = (S, A)$ qui représente le réseau routier d'une région, où le nombre situé sur l'arête joignant deux villes (ou sommets) est la distance en kilomètres entre ces deux villes. Les distances indiquées dans les rectangles sur le graphe ci-dessous représentent les distances à vol d'oiseau entre ce sommet et le sommet d'arrivée 8 (Pau).




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`.

**a)** Construire la matrice d'adjacence $\left(M_{i,j}\right)_{0\leq i,j\leq n-1}$ (appelée également **matrice de distance**) du graphe *G*, définie par : pour tous les indices *i, j*, $M_{i,j}$ 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=1e10` qui représente une distance infinie.

**b)** 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`]

**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`.

**Bouche while**

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 $f(\text{indice})$ 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=[172,156,140,217,164,72,67,128,0,92]` telle que `H[i]` = distance euclidienne entre le sommet *i* et le sommet `arrivée`=8.

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` (8). Cette fonction retourne le chemin suivi ainsi que la distance parcourue entre `départ` et `arrivée` en utilisant l'algorithme A*.

Écrire le programme principal permettant de déterminer un plus court chemin entre le sommet de départ 0 et le sommet d'arrivée 8. Le programme affichera le chemin suivi ainsi que la distance parcourue.

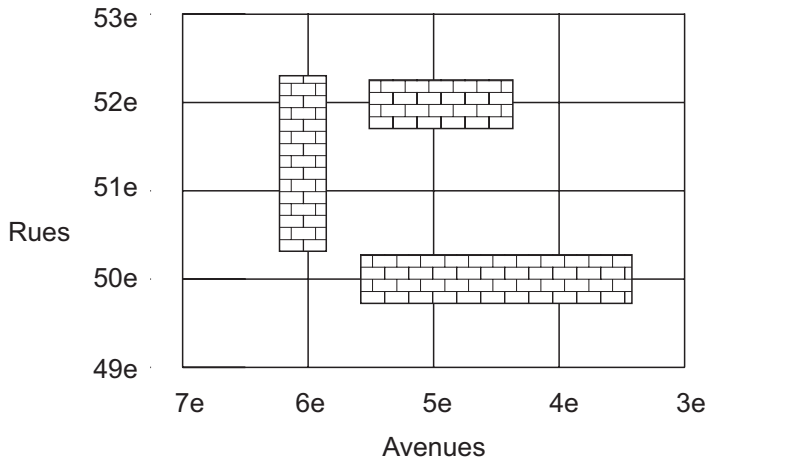
## 12.3

**Distance de Manhattan (sauf TSI et TPC)**

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=1e10` 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`.

**a)** Définir un dictionnaire `dico` représentant le graphe *G*. La clé associée à chaque sommet représente la liste des sommets adjacents.

**b)** 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`.

**c)** L'algorithme BFS glouton (*Best First Search* : meilleure première recherche) permet de déterminer un plus court chemin entre le sommet `départ` et le sommet `arrivée`.

**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`.

**Bouche while**

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.

**d)** 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 $f(i)=(1-w)g(i)+wh(i)$ :

- $g(i)$ 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`.

**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`.

**Bouche `while`**

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 $f(\text{indice})$ 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.

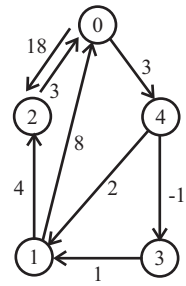
Écrire le programme principal permettant de déterminer un plus court chemin entre le sommet de départ 5 et le sommet d'arrivée 13. Le programme affichera le chemin suivi ainsi que la distance parcourue pour $w = 0$, $w = 0.5$ et $w = 1$.

**e)** Comment appelle-t-on les algorithmes lorsque $w = 0$, $w = 0.5$ et $w = 1$ ?

## 12.4

### Algorithme de Bellman-Ford (Spé, sauf TSI et TPC)

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 considère le graphe orienté fortement connexe $G = (S, A)$ constitué de $n$ sommets. Les arêtes sont orientées et le poids des arcs peut être négatif.




`L.reverse()` permet d'inverser les éléments de la liste `L`.

**a)** Construire la matrice d'adjacence $\left(M_{i,j}\right)_{0\leq i,j\leq n-1}$ du graphe $G$, définie par : pour tous les indices $i$, $j$, $M_{i,j}$ représente le poids de l'arc d'origine $i$ et d'extrémité $j$. Lorsque les sommets ne sont pas reliés, le poids de l'arc vaut l'infini. On définit la variable `inf=1e10` qui représente un poids infini.

**b)** On cherche à calculer la distance d'un plus court chemin entre le sommet `départ` et les autres sommets du graphe $G$ en utilisant l'algorithme de Bellman-Ford. On suppose dans cette question que le graphe n'a pas de cycle de poids négatif. On définit la liste `DIST` contenant $n$ éléments. `DIST[i]` désigne la distance du sommet `départ` au sommet $i$.

**Initialisation de l'algorithme de Bellman-Ford** :

- Toutes les distances sont estimées à `inf`.
- La distance du sommet `départ` au sommet `départ` vaut 0.

**Tant que la liste `DIST` est modifiée, on cherche une meilleure estimation de chaque élément de `DIST`** :

- Pour tous les arcs $(u, v)$ : si (distance de `départ` au sommet $u$ + $M[u][v]$ < distance de `départ` à $v$), alors on connaît une meilleure estimation de `DIST[v]`.

Écrire une fonction `Bellman1` qui admet comme arguments d'entrée la matrice d'adjacence `M` et le sommet `départ`. Cette fonction retourne la liste `DIST`.

**c)** Définir un cycle et un cycle de poids négatif.

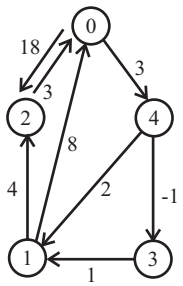
**d)** On cherche à mémoriser le chemin suivi et détecter si le graphe a un cycle de poids négatif.

- À chaque fois que l'on relâche l'arc $(u, v)$, c'est-à-dire que le chemin du sommet `départ` au sommet $v$ passe par le sommet $u$ : `PRECEDENT[v] = u`.

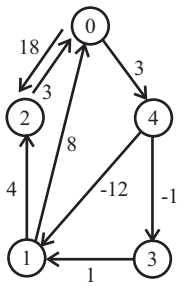
- On peut montrer que l'algorithme de Bellman-Ford (question b) s'arrête après au plus $n-1$ exécutions de la boucle « Tant que la liste `DIST` est modifiée » dans le cas d'un graphe sans cycle de poids négatif. Si un graphe a un cycle de poids négatif, alors il n'y a pas de plus court chemin depuis `début` vers les sommets qui sont sur ce cycle de poids négatif. La boucle « Tant que la liste `DIST` est modifiée » de la question b) s'exécuterait indéfiniment. En particulier, cette boucle s'exécuterait au moins $n$ fois. On peut donc exécuter la boucle $n$ fois sur un graphe quelconque et si, à la dernière itération, les éléments de `DIST` changent, alors il y a un cycle.

Écrire une fonction `Bellman2` qui admet comme arguments d'entrée la matrice d'adjacence `M`, le sommet de départ `départ` et le sommet d'arrivée `arrivée`. Cette fonction retourne `True, -inf, []` si le graphe possède un cycle de poids négatif. Sinon elle retourne `False`, la distance parcourue et le chemin suivi.

Écrire le programme principal permettant de déterminer un plus court chemin entre le sommet de départ 4 et le sommet d'arrivée 0 en utilisant la fonction `Bellman2`. Le programme affichera la distance parcourue ainsi que le chemin suivi. Le programme affichera également si le graphe possède un cycle de poids négatif.




Graphe n° 1




Graphe n° 2

Que renvoie le programme principal pour les graphes 1 et 2 ?

# Du mal à démarrer ?

## 12.1

**a)** Si deux sommets différents $i$ et $j$ sont reliés par une arête, alors $M[i][j]=M[j][i]=1$ sinon $M[i,j]=M[j][i]=0$.

**b)** Définir une boucle `while(position!=arrivée)`. Définir une boucle `for` pour parcourir toutes les villes non sélectionnées.

**c)** Compter le nombre d'opérations élémentaires dans le pire des cas.

## 12.2

**a)** Si deux sommets différents $i$ et $j$ sont reliés par une arête, alors $M[i][j]=M[j][i]=1$ sinon $M[i,j]=M[j][i]=0$.

**b)** Définir une boucle `while(position!=arrivée)`. Définir une boucle `for` pour parcourir toutes les villes non sélectionnées.

### 12.3

**a)** Le dictionnaire contient 25 clés. La valeur de chaque clé contient tous les sommets adjacents.

**b)** La distance de Manhattan entre un sommet $A$ (de coordonnées $x_A$, $y_A$) et un sommet $B$ (de coordonnées $x_B$, $y_B$) vaut $|x_B - x_A| + |y_B - y_A|$.

**c)** Définir une boucle `while(position!=arrivée)`. Définir une boucle `for` pour parcourir tous les sommets non sélectionnés.

**d)** Modifier la fonction d'évaluation.

### 12.4

**b)** Parcourir toutes les lignes et les colonnes de `M` pour ajouter les arcs i→j à la liste `ordre`. Parcourir tous les éléments (`elm`) de `ordre` et comparer `DIST[elm[1]]` à `DIST[elm[0]]+M[elm[0]][elm[1]]`.

**d)** Parcourir tous les éléments (`elm`) de `ordre` et comparer `DIST[elm[1]]` à :

`DIST[elm[0]]+M[elm[0]][elm[1]]`. Mettre à jour les listes `DIST` et `PRECEDENT`.

# Corrigés des exercices

### 12.1

**a)**

La matrice d'adjacence est : $M = \begin{pmatrix} 0 & 1 & 0,8 & 2,5 & \infty & \infty & \infty & \infty & \infty & \infty \\ 1 & 0 & \infty & \infty & 0,5 & 1,3 & 1,3 & \infty & \infty & \infty \\ 0,8 & \infty & 0 & 2,7 & \infty & \infty & \infty & 1 & 1,75 & \infty \\ 2,5 & \infty & 2,7 & 0 & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & 0,5 & \infty & \infty & 0 & \infty & \infty & \infty & \infty & \infty \\ \infty & 1,3 & \infty & \infty & \infty & 0 & 0,7 & \infty & 1,3 & 0,8 \\ \infty & 1,3 & \infty & \infty & \infty & 0,7 & 0 & \infty & 1,25 & \infty \\ \infty & \infty & 1 & \infty & \infty & \infty & \infty & 0 & 2,5 & \infty \\ \infty & \infty & 1,75 & \infty & \infty & 1,3 & 1,25 & 2,5 & 0 & 1,3 \\ \infty & \infty & \infty & \infty & \infty & 0,8 & \infty & \infty & 1,3 & 0 \end{pmatrix}$.

Comme le graphe n'est pas orienté, la matrice est symétrique.

**b)**

```
inf=1e10        # variable représentant l'infini

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):
    # 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=len(M)          # nombre de villes = nombre 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 # 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=[[0,1,0.8,2.5,inf,inf,inf,inf,inf,inf],\
   [1,0,inf,inf,0.5,1.3,1.3,inf,inf,inf],\
   [0.8,inf,0,2.7,inf,inf,inf,1,1.75,inf],\
   [2.5,inf,2.7,0,inf,inf,inf,inf,inf,inf],\
   [inf,0.5,inf,inf,0,inf,inf,inf,inf,inf],\
   [inf,1.3,inf,inf,inf,0,0.7,inf,1.3,0.8],\
   [inf,1.3,inf,inf,inf,0.7,0,inf,1.25,inf],\
   [inf,inf,1,inf,inf,inf,inf,0,2.5,inf],\
   [inf,inf,1.75,inf,inf,1.3,1.25,2.5,0,1.3],\
   [inf,inf,inf,inf,inf,0.8,inf,inf,1.3,0]]
départ, arrivée=0, 8        # ville de départ et ville d'arrivée
L, durée=dijkstra2(M, départ, arrivée)
print("TRAJET NUMERO 1")
print("Chemin suivi : ", L) # affichage de la liste des villes
print("Temps de trajet = ", durée, "h")
```

Le programme Python affiche :

- Chemin suivi : [0, 2, 8]
- Temps de trajet = 2.55 h

**Remarque**

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.

L'algorithme de Floyd-Warshall permet de traiter des arcs dont le poids est négatif (voir chapitre 13 « Programmation dynamique »).

**c)** Soit $n$ le nombre de villes.

Initialisation de la liste `VILLES` : une opération et, pour chaque valeur de $i$, une comparaison et une affectation. La boucle `for` est parcourue $n$ fois. On a donc $(2n + 1)$ opérations élémentaires.

Dans le pire des cas, on a $n - 1$ itérations dans la boucle `while`. Pour chaque valeur de `position`, on a :

- une comparaison, un calcul de somme, un test et deux affectations pour chaque valeur de $i$ : $5n$ opérations élémentaires ;
- une affectation : une opération élémentaire ;
- recherche du minimum : trois comparaisons et deux affectations pour chaque valeur de $i$ : $5n$ opérations élémentaires ;
- deux affectations : deux 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.

## 12.2

**a)** La matrice d'adjacence est : $M = \begin{pmatrix} 0 & 45 & 50 & 206 & \infty & \infty & \infty & \infty & \infty & \infty \\ 45 & 0 & \infty & \infty & 24 & 125 & 107 & \infty & \infty & \infty \\ 50 & \infty & 0 & 170 & \infty & \infty & \infty & 97 & 167 & \infty \\ 206 & \infty & 170 & 0 & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & 24 & \infty & \infty & 0 & \infty & \infty & \infty & \infty & \infty \\ \infty & 125 & \infty & \infty & \infty & 0 & 54 & \infty & 93 & 52 \\ \infty & 107 & \infty & \infty & \infty & 54 & 0 & \infty & 96 & \infty \\ \infty & \infty & 97 & \infty & \infty & \infty & \infty & 0 & 175 & \infty \\ \infty & \infty & 167 & \infty & \infty & 93 & 96 & 175 & 0 & 115 \\ \infty & \infty & \infty & \infty & \infty & 52 & \infty & \infty & 115 & 0 \end{pmatrix}$.

Comme le graphe est orienté, la matrice n'est pas nécessairement symétrique.

**b)** 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.

**Initialisation de l'algorithme A***

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`.

**Bouche `while`**

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 $f(\text{indice})$ 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.

**Remarque**

On utilise la distance à vol d'oiseau dans cet exercice alors que dans l'exercice 12.3 « 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.

```
inf=1e10                            # variable représentant l'infini

def init3(départ, n):               # n est le 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 car sommet de départ
                # 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 un 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=len(M)          # nombre de sommets = nombre de lignes de M
    inf=1e10          # variable représentant l'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]

# initialisation du programme
M=[[0,45,50,206,inf,inf,inf,inf,inf,inf],\
   [45,0,inf,inf,24,125,107,inf,inf,inf],\
   [50,inf,0,170,inf,inf,inf,97,167,inf],\
   [206,inf,170,0,inf,inf,inf,inf,inf,inf],\
   [inf,24,inf,inf,0,inf,inf,inf,inf,inf],\
   [inf,125,inf,inf,inf,0,54,inf,93,52],\
   [inf,107,inf,inf,inf,54,0,inf,96,inf],\
   [inf,inf,97,inf,inf,inf,inf,0,175,inf],\
   [inf,inf,167,inf,inf,93,96,175,0,115],\
   [inf,inf,inf,inf,inf,52,inf,inf,115,0]]
départ, arrivée=0, 8
H=[172, 156, 140, 217, 164, 72, 67, 128, 0, 92]
L2, dist2=algoA(M, H, départ, arrivée)
print()
print('Algo A* chemin :',L2,'; distance =',dist2, "km")
```

Le programme Python affiche :

- Algo A* chemin : [0, 2, 8]
- distance = 217 km

**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. Les algorithmes de Bellman-Ford et Floyd-Warshall (voir chapitre 13 « Programmation dynamique ») permettant de traiter des arcs dont le poids est négatif.

## 12.3

**a)**

```
dico={0:[1,5], 1 :[0,2], 2 :[1,3], 3 :[2,4,8], 4 :[3,9],
5 :[0,10], 6 :[], 7 :[], 8 :[3,9,13], 9 :[4,8,14],
10 :[5,15], 11 :[], 12 :[13], 13 :[8,12,14], 14 :[9,13,19],
15 :[10,16,20], 16 :[15,21], 17 :[], 18 :[], 19 :[14,24],
20 :[15,21,], 21 :[16,20,22], 22 :[21,23], 23 :[22,24], 24 :[19,23]}
```

Le sommet 3 est relié aux sommets 2, 4 et 8. La valeur de la clé 3 est la liste des sommets adjacents [2, 4, 8]. Le nombre d'élément 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 :

```
import math as m   # module math renommé m
nb_rues=int(m.sqrt(n))
           # n = nombre de sommets = nombre de rues * nombre d'avenues
```

**b)** La distance de Manhattan entre un sommet $A$ (de coordonnées $x_A$, $y_A$) et un sommet $B$ (de coordonnées $x_B$, $y_B$) 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.

**Remarque**

Dans l'exercice 12.2 « Algorithme A* », on utilise la distance euclidienne (ou distance à vol d'oiseau).

```
inf=1e10                    # variable représentant l'infini

def listeH(arrivée, n):
    import math as m        # module math renommé m
    # 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(m.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=abs(yB-yA)+abs(xB-xA)        # distance de Manhattan pour le sommet i
        H.append(y)
    return H
```

**c)** 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.

**Initialisation de l'algorithme**

On définit la liste `H` en utilisant la fonction `listeH` définie dans la question b).

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`.

**Bouche `while`**

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 $f(\text{indice})$ 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 est le 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 car sommet de départ
                # 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 un 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=1e10                           # variable représentant l'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]
```

**d)** 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 b) 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 compris entre 0 et 1.
```

```
    # On récupère la liste SOMMETS
    n=len(dico)                         # nombre de sommets
    H=listeH(arrivée, n)
    inf=1e10                            # variable représentant l'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 départ->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
        # 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=5, 13              # sommet de départ et sommet d'arrivée
L2, dist2=BFS(dico, départ, arrivée)
print('Algo BFS : chemin :', L2, '; distance =', dist2, "km")
w=0
```

```
for i in range(3):
    L2, dist2=MANHATTAN(dico, départ, arrivée, w)
    print('w =',w, '; chemin :', L2, '; distance =', dist2, "km")
    w=w+0.5
```

**e)** 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 BFS glouton.

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` = 5 et `arrivée` = 13.

- $w = 0$ ; chemin : [5, 0, 1, 2, 3, 8, 13] ; distance = 6 km.
- $w = 0.5$ ; chemin : [5, 0, 1, 2, 3, 8, 13] ; distance = 6 km.
- $w = 1.0$ ; chemin : [5, 10, 15, 16, 21, 22, 23, 24, 19, 14, 13] ; distance = 10 km

L'algorithme BFS ($w = 1$) ne donne pas la solution optimale : il privilégie le chemin passant par 10, qui semble plus près de 0 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 qu'avec la programmation dynamique on peut revenir sur les choix précédents. L'algorithme est glouton pour toutes les valeurs de $w$.

## 12.4

**a)** La matrice d'adjacence est : $M = \begin{pmatrix} 0 & \infty & 18 & \infty & 3 \\ 8 & 0 & 4 & \infty & \infty \\ 3 & \infty & 0 & \infty & \infty \\ \infty & 1 & \infty & 0 & \infty \\ \infty & 2 & \infty & -1 & 0 \end{pmatrix}$.

Comme le graphe est orienté, la matrice n'est pas nécessairement symétrique.

**b)** Le booléen `RELACHE` permet de savoir si la liste `DIST` a été modifiée.

Si (distance de `départ` au sommet $u$ + distance de $u$ à $v$) < distance de `départ` à $v$, alors on connaît une meilleure estimation de `DIST[v]`. On relâche l'arc $(u, v)$, c'est-à-dire que le chemin de `départ` à $v$ passe par le sommet $u$. Le booléen `RELACHE` prend alors la valeur `True`.

- Dans l'algorithme de Dijkstra (voir exercice 12.1 « Algorithme de Dijkstra »), on calcule la distance de `départ` à tous les sommets $i$ non sélectionnés en passant par le sommet `position`. On cherche ensuite le minimum des distances pour les villes non sélectionnées. Ce minimum permet de déclarer un sommet sélectionné. Cette distance est définitive.
- Dans l'algorithme de Bellman-Ford, on peut trouver ultérieurement une distance encore plus petite en passant par d'autres sommets avec des arêtes de poids négatif. Dès que la liste `DIST` est modifiée, la

variable `RELACHE` passe à `True`. Il faudra recalculer toutes les distances à la prochaine étape de la boucle `while RELACHE==True:`.

La ligne `DIST[elm[1]] = DIST[elm[0]]+M[elm[0]][elm[1]]` met à jour la liste `DIST`. On dit que l'on relâche l'arc (*u*, *v*), c'est-à-dire que le chemin de `début` à *v* passe par le sommet *u*.

```
inf=1e10                            # variable représentant l'infini


def Bellman(M, départ):
    # la fonction retourne la liste DIST pour la matrice d'adjacence M
    # et le sommet départ (int).
    # DIST[i] = distance du sommet départ au sommet i
    n=len(M)                        # nombre de lignes de M
    ordre=[]                        # initialisation de la liste des arcs
    for i in range(n):              # parcourt toutes les lignes de M
        for j in range(n):          # parcourt toutes les colonnes de M
            if M[i][j]!=0 and M[i][j]!=inf:
                ordre.append([i,j])  # ajoute l'arc i->j
    DIST=[inf]*n     # toutes les distances de la liste = inf
    DIST[départ]=0   # distance nulle de départ à départ
    RELACHE=True     # booléen permettant de savoir si DIST est modifiée
    while RELACHE==True:
        RELACHE=False
        for elm in ordre:           # pour tous les arcs (u, v)
            if DIST[elm[1]] > DIST[elm[0]]+M[elm[0]][elm[1]]:
                DIST[elm[1]] = DIST[elm[0]]+M[elm[0]][elm[1]]
                RELACHE=True        # on relâche l'arc
                        # la liste DIST est modifiée
    return DIST

M=[[0, inf, 18, inf, 3],
   [8, 0, 4, inf, inf],
   [3, inf, 0, inf, inf],
   [inf, 1, inf, 0, inf],
   [inf, 2, inf, -1, 0]]
départ=4
DIST=Bellman(M, départ)
print("Liste des distances :", DIST)
```

**c)** Un cycle est un chemin simple tel que le sommet d'arrivée est le même que le sommet de départ.

Un cycle de poids négatif (ou absorbant) est un cycle pour lequel la somme des poids des arcs est négative.

**d)** On définit les listes suivantes contenant *n* éléments : `DIST`, `PRECEDENT`.

- `DIST[i]` : distance du sommet `départ` au sommet *i*.
- `PRECEDENT[i]` : sommet qui précède *i* lors du chemin de `départ` au sommet *i*.

Initialisation :

- Toutes les distances sont estimées à `inf`.
- Tous les éléments de la liste `PRECEDENT` valent −1.
- La distance du sommet `départ` au sommet `départ` vaut 0.

Tant que la liste `DIST` est modifiée (boucle `while` exécutée $n - 1$ fois au maximum), on cherche une meilleure estimation de chaque élément de `DIST` :

- Pour tous les arcs ($u$, $v$) : Si `DIST[u] + M[u][v] < DIST[v]`, alors `DIST[u] + M[u][v]` est une meilleure estimation de `DIST[v]` et `PRECEDENT[v]=u`. On relâche l'arc ($u$, $v$), c'est-à-dire que le chemin de `début` à $v$ passe par le sommet $u$.

Test pour savoir si le graphe possède un cycle de poids négatif :

- Pour tous les arcs ($u$, $v$) :
  - Si `DIST[u] + M[u][v] < DIST[v]` :
    alors le graphe possède un cycle de poids négatif.

```
def Bellman2(M, départ, arrivée):
    # la fonction retourne la distance parcourue et le chemin suivi
    # entre départ (int) et arrivée (int) pour la matrice d'adjacence M
    n=len(M)                        # nombre de lignes de M
    ordre=[]                        # initialisation de la liste des arcs
    for i in range(n):              # i varie entre 0 inclus et n exclu
        for j in range(n):          # j varie entre 0 inclus et n exclu
            if M[i][j]!=0 and M[i][j]!=inf:
                ordre.append([i,j]) # ajoute l'arc i->j

    DIST=[inf]*n                    # toutes les distances de la liste = inf
    PRECEDENT=[-1]*n                # tous les sommets précédents = -1
    DIST[départ]=0                  # distance nulle de départ à départ
    i=1
    RELACHE=True
    while (i<=n-1) and RELACHE==True:
        RELACHE=False               # relaxation des sommets
        for elm in ordre:           # pour tous les arcs (u, v)
            if DIST[elm[1]] > DIST[elm[0]]+M[elm[0]][elm[1]]:
                DIST[elm[1]] = DIST[elm[0]]+M[elm[0]][elm[1]]
                PRECEDENT[elm[1]] = elm[0]
                RELACHE=True        # on relache l'arc
        i+=1

    # teste si le graphe possède un cycle de poids négatif
    RELACHE=False
    for elm in ordre:               # pour tous les arcs (u, v)
        if DIST[elm[1]] > DIST[elm[0]]+M[elm[0]][elm[1]]:
            RELACHE=True            # on relâche l'arc
    if RELACHE==True:
            # le graphe possède un cycle de poids négatif
        return True, -inf, []
    else:   # le graphe ne possède pas de cycle de poids négatif
        i=arrivée
        chemin=[arrivée]   # initialisation de la liste des sommets parcourus
        while (i!=départ):
```

```
            i=PRECEDENT[i]
            chemin.append(i)
                # on obtient la liste des sommets en sens inverse
        chemin.reverse()            # il faut inverser la liste chemin
        return False, DIST[arrivée], chemin

matriceAdj1 = [[0, inf, 18, inf, 3],
               [8, 0, 4, inf, inf],
               [3, inf, 0, inf, inf],
               [inf, 1, inf, 0, inf],
               [inf, 2, inf, -1, 0]]

matriceAdj2 = [[0, inf, 18, inf, 3],
               [8, 0, 4, inf, inf],
               [3, inf, 0, inf, inf],
               [inf, 1, inf, 0, inf],
               [inf, -12, inf, -1, 0]]
M1=matriceAdj1   # matrice1 d'adjacence
départ=4                            # sommet de départ
arrivée=0                           # sommet d'arrivée
flag1, distance1, chemin1=Bellman2(M1, départ, arrivée)
if flag1==True:
    print("Le graphe n°1 possède un cycle de poids négatif.")
else:
   print("Le graphe n°1 ne possède pas de cycle de poids négatif.")
   print("Distance n°1 parcourue :", distance1)
   print("Chemin n°1 :", chemin1)

M2=matriceAdj2                      # matrice2 d'adjacence
départ=4                            # sommet de départ
arrivée=0                           # sommet d'arrivée
flag2, distance2, chemin2=Bellman2(M2, départ, arrivée)
if flag2==True:
    print("Le graphe n°2 possède un cycle de poids négatif.")
else:
    print("Le graphe n°2 ne possède pas de cycle de poids négatif.")
    print("Distance n°2 parcourue :", distance2)
    print("Chemin n°2 :", chemin2)
```

On a deux conditions pour la boucle `while: i<=n-1` d'après l'énoncé et `RELACHE==True` (la liste `DIST` a été modifiée).

Les deux matrices d'adjacence sont : $M_1 = \begin{pmatrix} 0 & \infty & 18 & \infty & 3 \\ 8 & 0 & 4 & \infty & \infty \\ 3 & \infty & 0 & \infty & \infty \\ \infty & 1 & \infty & 0 & \infty \\ \infty & 2 & \infty & -1 & 0 \end{pmatrix}$ et $M_2 = \begin{pmatrix} 0 & \infty & 18 & \infty & 3 \\ 8 & 0 & 4 & \infty & \infty \\ 3 & \infty & 0 & \infty & \infty \\ \infty & 1 & \infty & 0 & \infty \\ \infty & -12 & \infty & -1 & 0 \end{pmatrix}$.

Pour le **graphe n° 1**, le programme Python affiche :

- Le graphe n°1 ne possède pas de cycle de poids négatif.
- Distance n°1 parcourue : 7.0
- Chemin n°1 : [4, 3, 1, 2, 0]

Pour le **graphe n° 2**, le programme Python affiche :

- Le graphe n°2 possède un cycle de poids négatif.

# Programmation dynamique (spé, sauf TSI et TPC)

CHAPITRE 13

Plan



## Thèmes abordés dans les exercices

- Méthode « diviser pour régner »
- Chevauchement de sous-problèmes
- Méthode gloutonne
- Programmation dynamique
- Techniques *Top Down* (de haut en bas), *Bottom Up* (de bas en haut)
- Utilisation d'un dictionnaire pour la mémoïsation

## Points essentiels du cours pour la résolution des exercices

- Reconnaître un choix qui semble le meilleur sur le moment
- Comparer la méthode gloutonne et la programmation dynamique
- Décomposer le problème en sous-problèmes traités récursivement
- Obtenir la solution optimale au problème en utilisant une récurrence
- Nécessité de traiter tous les sous-problèmes avec la méthode *Bottom Up*
- Reconstruction d'une solution optimale à partir de l'information calculée

# Les méthodes à retenir

Un loueur de skis cherche à optimiser la location de $m$ paires de skis à $n$ skieurs $(m \geq n)$, c'est-à-dire fournir à chaque client une paire de skis dont la taille est au plus proche de la taille du client. On utilise deux listes `L` et `S` :

- La taille du $i^{\text{ème}}$ skieur est stockée dans *L[i]*.
- La longueur de la $j^{\text{ième}}$ paire de skis est stockée dans `S[j]`.

On suppose que les listes `L` et `S` sont triées par ordre croissant. On pose :

```
L=[0, 140, 150, 150, 162]        # liste des 4 skieurs de taille 140, 150, 150, 162
S=[0, 131, 148, 153, 160, 180]   # liste des 5 paires de skis
                                 # de longueur 131, 148, 153, 160, 180
```

On définit une liste d'attribution des paires de skis pour les $n$ skieurs : `F[i]` désigne l'indice du ski attribué au $i^{\text{ème}}$ skieur. Le problème consiste à trouver une liste d'attribution `F` telle que le $i^{\text{ème}}$ skieur se voie attribuer la paire de skis `F[i]` qui minimise la fonction d'évaluation : $\sum_{i=0}^{n-1} \left| S\left[F[i]\right] - L[i] \right|$.

# Méthode gloutonne

On utilise la méthode intuitive consistant à parcourir la liste des skieurs et à choisir la paire de skis optimale pour chaque skieur.

```
inf=1e10                          # variable représentant l'infini
def loueur_glouton(n, m):         # méthode gloutonne
    total=0
    T=[0 for i in range(m+1)]     # 1 si paire de skis déjà attribué, 0 sinon
    F=[0 for i in range(n+1)]
    for i in range(1, n+1):       # i varie entre 1 inclus et n inclus
        mini=inf
        indice=0
        for j in range(1, m+1):
            if abs(S[j]-L[i])<mini and T[j]==0:  # paire de skis j non attribué
                mini=abs(S[j]-L[i])
                indice=j
        total=total+abs(S[indice]-L[i])
        F[i]=indice
        T[indice]=1   # on a attribué la paire de skis de numéro indice
            # on ne revient plus sur ce choix -> méthode gloutonne
    return total, F

L=[0,140,150,150,162]             # liste des skieurs
S=[0,131,148,153,160,180]         # liste des paires de skis
n=len(L)-1                        # nombre de skieurs
m=len(S)-1                        # nombre de paires de skis
resultat1, F=loueur_glouton(n, m)
print(resultat1, F)
```

La fonction `loueur_glouton` retourne la fonction d'évaluation et la liste d'attribution `F` correspondant aux `n` premiers skieurs et `m` premières paires de skis. La liste `F` vaut [0, 2, 3, 4, 5].

Skieur de taille 140 → paire de skis de longueur 148

Skieur de taille 150 → paire de skis de longueur 153

Skieur de taille 150 → paire de skis de longueur 160

Skieur de taille 162 → paire de skis de longueur 180

La fonction d'évaluation vaut $\text{resultat1} = |148-140| + |153-150| + |160-150| + |180-162| = 39$.

Le problème est que l'on a fait un mauvais choix pour le premier ski. On ne revient plus sur ce choix. On pourra résoudre ce problème avec la programmation dynamique.

Cette méthode est appelée méthode gloutonne car elle consiste à faire le meilleur choix sur le moment, c'est-à-dire attribuer la paire de skis qui a la longueur la plus proche de la taille du $i^{\text{ème}}$ skieur.

## Programmation dynamique

La programmation dynamique est souvent utilisée pour résoudre des problèmes d'optimisation. Elle comprend plusieurs étapes :

- Recherche d'une récurrence pour déterminer la valeur d'une solution optimale.
- Utilisation de la technique *Top Down* (de haut en bas) ou *Bottom Up* (de bas en haut).

La programmation dynamique et la méthode gloutonne reposent sur l'utilisation de sous-problèmes. Il y a une différence importante :

- Dans la programmation dynamique, on calcule toutes les solutions des sous-problèmes, que l'on combine pour obtenir une solution optimale.
- Dans la méthode gloutonne, on choisit une solution qui semble être la meilleure et on résout le sous-problème qui en résulte.

On rencontre deux techniques dans la programmation dynamique :

- Technique récursive « *Top Down* » de mémoïsation : on résout dans le sens des données de grande taille vers les données de petite taille. Lors d'un appel récursif, on regarde dans une liste intermédiaire si le sous-problème est déjà traité.
- Technique itérative « *Bottom Up* » : on résout dans le sens des données de petite taille vers les données de grande taille (c'est l'ordre inverse de « *Top Down* »). On stocke également les résultats obtenus dans une liste intermédiaire.

On définit $A[n][m]=\sum_{i=0}^{n-1}\left|S\left[F[i]\right]-L[i]\right|$ où `F` est la meilleure attribution possible du problème restreint aux `n` premiers skieurs et `m` premières paires de skis. On initialise à l'infini (variable `inf`) tous les éléments de `A` qui n'ont pas encore été traités.

Pour trouver la solution optimale au problème du loueur de skis, on considère le principe suivant pour définir `A[n][m]` :

- Si `A[n][m]` < `inf`, on a déjà calculé la fonction d'évaluation optimale. Il suffit de retourner cette valeur. Le sous-problème est déjà résolu. C'est une condition d'arrêt de la fonction récursive.
- Si $n = 0$ ou $m = 0$. On considère 0 skieur ou 0 paire de skis. La fonction d'évaluation vaut : `A[n][m]=0`. Le sous-problème est déjà résolu. C'est une condition d'arrêt de la fonction récursive.
- Si $n > m$, il y a trop de skieurs par rapport au nombre de paires de skis. La fonction d'évaluation vaut : `A[n][m]=inf`. Le sous-problème est déjà résolu. C'est une condition d'arrêt de la fonction récursive.
- Si les conditions précédentes ne sont pas réunies, on calcule récursivement `A[n][m-1]` et `A[n-1][m-1]`. On a alors $A[n][m]=\min\left(\left(A[n][m-1],\right)\left(A[n-1][m-1]+\left|S[m]-L[n]\right|\right)\right)$.

```
def loueur_dyn(A, n, m):        # programmation dynamique top down
    if A[n][m]<inf:             # condition d'arrêt
        return A[n][m]
    else:
        if n==0 or m==0:        # condition d'arrêt avec 0 skieur ou 0 ski
            A[n][m]=0
            return 0
```

```
        elif n>m: # trop de skieurs par rapport au nombre de paires de skis
            return inf          # condition d'arrêt - valeur infinie
        else:
            a=loueur_dyn(A, n, m-1)
            b=loueur_dyn(A, n-1, m-1)+abs(S[m]-L[n])
            if a<b and m>=2:    # il faut avoir au moins deux paires
                                # de skis pour tester m-1
                mini=a
            else:
                mini=b
            A[n][m]=mini
            return mini

A=[[inf for j in range(m+1)] for i in range(n+1)]
       # A[i, j]    i : skieur et j : paire de skis
resultat3=loueur_dyn(A, n, m)
print(resultat3)
```

La fonction d'évaluation vaut `resultat3=16`.

La méthode « diviser pour régner » peut se décomposer en trois étapes :

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

L'algorithme est de type « diviser pour régner » puisqu'on décompose le problème (calcul de `A[n][m]`) en plusieurs sous-problèmes : calcul de `A[n][m-1]`, `A[n-1][m-1]`. On combine les différents sous-problèmes pour résoudre le problème de départ. L'algorithme est bien acyclique car `m-1` < `m` et `n-1` < `n`. On traite bien des sous-problèmes que l'on peut traiter de façon optimale.

**Remarques**

- `A[n][m]` retourne la fonction d'évaluation optimale du problème restreint aux `n` premiers skieurs et `m` premières paires de skis. On stockera au fur et à mesure `A[n][m]`. On n'a pas besoin de recalculer `A[n][m]` s'il a déjà été calculé.
- Il faut bien définir les conditions d'arrêt de la fonction récursive : $n = 0$, $m = 0$ et $n > m$.

La fonction `loueur_dyn` utilise la programmation dynamique avec la technique « *Top Down* » de mémoïsation et permet d'obtenir une solution optimale sans résoudre plusieurs fois le même sous-problème. Voir exercice 13.3 « Loueur de skis » pour l'utilisation de la technique *Bottom Up*.

# Énoncés des exercices

## 13.1

■■□□

**Suite des nombres de Fibonacci, *Top Down* et *Bottom Up* (sauf TSI et TPC)**

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

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

**a)** 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 » ?

**b)** Représenter l'arbre des appels de la fonction récursive `fibo1(5)`. Combien de fois est recalculé $F_2$ ? Quel est l'inconvénient ?

Pour pallier cet inconvénient, on utilise deux techniques : technique récursive « *Top Down* » (de haut en bas) de mémoïsation et technique itérative « *Bottom Up* » (de bas en haut).

**c)** La technique de mémoïsation consiste à stocker les valeurs de $F_n$ dans une liste au fur et à mesure qu'elles sont calculées. Utiliser un dictionnaire pour implémenter la mémoïsation. Écrire une fonction « récursive » `fibo2` renvoyant le nombre de Fibonacci $F_n$ en utilisant la technique « *Top Down* ».

**d)** Écrire une fonction itérative `fibo3` qui prend en argument un entier naturel $n$ et renvoie le nombre de Fibonacci $F_n$ en utilisant la technique « *Bottom Up* ».

## 13.2

■■■□

**Découpe d'une barre de métal (sauf TSI et TPC)**

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]]`.

**a)** 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 retourne le prix optimal que l'on peut obtenir de cette barre.

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 ?

**b)** On dispose d'une liste $P$ des prix de vente des barres en fonction de leur longueur : $P[i]$ désigne le prix de vente d'une barre de longueur $i \in [\![0, n]\!]$. `P = [0, 3, 5, 14, 16, 22]` désigne la liste des prix de vente pour une barre de longueur 5. On appelle $M(n)$ le prix optimal que l'on obtenir d'une barre de longueur $n$. Pour trouver la solution optimale au problème de la découpe, on utilise le principe suivant :

$$M(n) = \max\left(p(n), \max_{0<i<n}\left(M(i) + M(n-i)\right)\right).$$

Écrire une fonction récursive `decoupe2` qui admet comme arguments une liste `P` et un entier `n`. La fonction retourne le prix optimal que l'on obtenir de cette barre.

L'algorithme utilise-t-il la méthode « diviser pour régner » ? Quel est le principal inconvénient de cet algorithme ?

**c)** Écrire une fonction récursive `decoupe3` qui admet comme arguments une liste `P`, un entier `j` (barre de longueur `j`) et une liste `L` (stockage des résultats intermédiaires : `L[i]` = prix optimal d'une barre de longueur `i`). La fonction retourne le prix optimal que l'on peut obtenir de cette barre en utilisant la programmation dynamique. Utilise-t-on la technique « *Top Down* » (de haut en bas) ou « *Bottom Up* » (de bas en haut) ?

**d)** Écrire une fonction itérative `decoupe4` qui admet comme arguments une liste `P`, un entier `n` et une liste `L` (stockage des résultats intermédiaires `L[i]` = prix optimal d'une barre de longueur `i`). La fonction retourne le prix optimal que l'on peut obtenir de cette barre en utilisant la programmation dynamique. Les résultats intermédiaires sont stockés dans la liste `L`.

On calcule les éléments de `L` en partant de la plus petite valeur de $n$. Utilise-t-on la technique « *Top Down* » ou « *Bottom Up* » ? Quelle est la principale différence entre ces deux techniques en comparant les listes `L` obtenues par les deux algorithmes précédents ?

**e)** On souhaite reconstruire la solution optimale à partir de l'information calculée, c'est-à-dire connaître la liste des prix des différentes découpes de la barre de longueur $n$. On définit la liste `T` :

- `T[m][i]` = 1 si on a découpé à l'abscisse $i$ une barre de longueur $m$ ;
- `T[m][i]` = 0 sinon.

On définit une liste `ABSCISSES` initialement vide. On pose $m = n$. On considère une boucle `while m>0` :

- Si `T[m][i]` = 1, alors on a découpé à l'abscisse $i$ la barre de longueur $m$. Ajouter cette abscisse dans la liste `ABSCISSES` et retrancher cette abscisse à la longueur $m$ de la barre.

Modifier la fonction `decoupe3` qui utilise la technique « *Top Down* » pour qu'elle retourne le prix optimal et la liste des abscisses des découpes.

**f)** Modifier la fonction `decoupe4` qui utilise la technique « *Bottom Up* » pour qu'elle retourne le prix optimal et la liste des abscisses des découpes.

## 13.3

### Loueur de skis (sauf TSI et TPC)

Un loueur de skis cherche à optimiser la location de $m$ paires de skis à $n$ skieurs $(m \geq n)$, c'est-à-dire fournir à chaque client une paire de skis dont la taille est au plus proche de la taille du client. On utilise deux listes `L` et `S` :

- La taille du $i^{\text{ème}}$ skieur est stockée dans `L[i]`.
- La longueur de la $j^{\text{ième}}$ paire de skis est stockée dans `S[j]`.

On suppose que les listes `L` et `S` sont triées par ordre croissant. On pose `L[0]=0` et `S[0]=0`. Par exemple, on a :

```
L=[0, 140, 150, 150, 162]          # liste des 4 skieurs de taille 140, 150, 150, 162
S=[0, 131, 148, 153, 160, 180]     # liste des 5 paires de skis
                                   # de longueur 131, 148, 153, 160, 180
```

On définit une liste d'attribution des paires de skis pour les $n$ skieurs : `F[i]` désigne l'indice du ski attribué au $i^{\text{ème}}$ skieur.

Le problème consiste à trouver une liste d'attribution `F` telle que le $i^{\text{ème}}$ skieur se voie attribuer la paire de skis `F[i]` qui minimise la fonction d'évaluation : $\sum_{i=0}^{n-1} \left| S\left[F[i]\right] - L[i] \right|$.

**a)** On utilise la méthode intuitive consistant à parcourir la liste des skieurs et à choisir la paire de skis optimale pour chaque skieur. Écrire une fonction itérative `optim1` qui admet comme arguments deux entiers `n` et `m`. Cette fonction retourne la fonction d'évaluation et la liste d'attribution `F` correspondant aux `n` premiers skieurs et `m` premières paires de skis. Pourquoi cette méthode est-elle appelée gloutonne ? Est-ce que `optim1(n, m)` retourne la fonction d'évaluation optimale ?

**b)** On définit $A[n][m] = \sum_{i=0}^{n-1} \left| S\left[F[i]\right] - L[i] \right|$ où `F` est la meilleure attribution possible du problème restreint aux `n` premiers skieurs et `m` premières paires de skis. Pour trouver la solution optimale au problème du loueur de skis, on considère le principe suivant pour définir `A[n][m]` :

- On suppose qu'un skieur sans paire de skis correspond au cas où la paire de skis est de taille nulle. On pose : `A[n][0]=0`. De même, s'il n'y a aucun skieur : `A[0][m]=0`.
- Si $n > m$, il y a trop de skieurs par rapport au nombre de paires de skis, alors `A[n][m]=inf`.
- On calcule récursivement `A[n][m-1]` et `A[n-1][m-1]`. On a alors $A[n][m] = \min A[n][m-1]$, $\left(A[n-1][m-1] + \left|S[m] - L[n]\right|\right)$. $A[n][m] = A[n][m-1]$ correspond à la paire de skis `m` associée à aucun skieur et $A[n][m] = A[n-1][m-1] + \left|S[m] - L[n]\right.$ correspond à la paire de skis `m` associée au skieur `n`.

Écrire une fonction récursive `optim2` qui admet comme arguments deux entiers `n`, `m` et retourne la fonction d'évaluation optimale du problème restreint aux `n` premiers skieurs et `m` premières paires de skis.

L'algorithme utilise-t-il la méthode « diviser pour régner » ? Quel est le principal inconvénient de cet algorithme ?

**c)** Écrire une fonction récursive `optim3` qui admet comme arguments une liste `A`, deux entiers `n`, `m` et retourne la fonction d'évaluation optimale du problème restreint aux `n` premiers skieurs et `m` premières paires de skis. On stocke dans la liste `A` les résultats intermédiaires : `A[i][j]` = fonction d'évaluation du problème restreint aux `i` premiers skieurs et `j` premières paires de skis. Toutes les valeurs de la liste `A` sont initialisées à `inf=1e10`. Utilise-t-on la technique « *Top Down* » ou « *Bottom Up* » ?

**d)** Écrire une fonction itérative `optim4` qui admet comme arguments une liste `A`, deux entiers `n` et `m`. La fonction retourne la fonction d'évaluation optimale du problème restreint aux `n` premiers skieurs et `m` premières paires de skis. On stocke dans la liste `A` les résultats intermédiaires : `A[i][j]` = fonction d'évaluation du problème restreint aux `i` premiers skieurs et `j` premières paires de skis. On calcule les éléments de `A` en partant des plus petites valeurs de `i` et `j`.

Utilise-t-on la technique « *Top Down* » ou « *Bottom Up* » ?

Quelle est la principale différence entre ces deux techniques en comparant les listes `A` obtenus par les deux algorithmes précédents ?

**e)** On souhaite reconstruire la solution optimale à partir de l'information calculée, c'est-à-dire connaître la liste d'attribution `F` des paires de skis. On parcourt la solution de la fin vers le début en reconstruisant les choix. On définit la liste `T` à deux dimensions :

- `T[i][j]=1` si on attribue la $j^{\text{ième}}$ paire de skis au $i^{\text{ème}}$ skieur.
- `T[i][j]=0` sinon.

On définit une liste `F` initialement vide. On pose `j=m`. On considère une boucle `for` en partant de $i = n$ :

- On considère une boucle `while` en partant de `j`. On fait décroître `j` jusqu'à trouver une valeur non nulle pour `T[i][j]`.
- Si on trouve une valeur non nulle pour `T[i][j]`, alors `F[i]=j` et on quitte la boucle `while`.
- `j` décroît d'une unité.

Écrire le programme principal en modifiant la fonction `optim3` pour obtenir la fonction d'évaluation optimale ainsi que la liste d'attribution `F`.

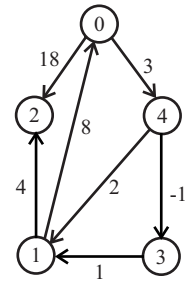
## 13.4

**Algorithme de Floyd-Warshall – Programmation dynamique (sauf TSI et TPC)**

On considère le graphe orienté $G = (S, A)$ constitué de $n$ sommets. Les arêtes sont orientées et le poids des arcs peut être négatif. On suppose que le graphe n'a pas de cycle de poids négatif. On définit la matrice d'adjacence $\left(M_{i,j}\right)_{0 \le i,j \le n-1}$ du graphe. L'algorithme de Floyd-Warshall définit $d_k(i,j)$ la distance minimale d'un chemin du sommet $i$ au sommet $j$ en empruntant des sommets intermédiaires d'indice strictement inférieur à $k$.

- Si $k = n$, alors $d_n(i,j)$ est la plus courte distance entre $i$ et $j$.
- Un chemin qui emprunte des sommets intermédiaires d'indice strictement inférieur à 0 ne peut emprunter aucun sommet intermédiaire, donc $d_0(i,j) = M[i][j]$.
- On considère un chemin optimal de $i$ à $j$ qui emprunte des sommets intermédiaires d'indice strictement inférieur à $k$. On a deux possibilités :
  - soit ce chemin ne passe jamais par le sommet $k$–1,
  - soit ce chemin passe exactement une fois par le sommet $k$–1.

  $d_k(i,j) = \min\left(d_{k-1}(i,j), d_{k-1}(i,k-1) + d_{k-1}(k-1,j)\right)$




On utilise les listes de listes pour représenter les matrices dans Python.

`L.reverse()` permet d'inverser les éléments de la liste `L`.

**a)** Construire la matrice d'adjacence $\left(M_{i,j}\right)_{0 \le i,j \le n-1}$ du graphe $G$, définie par : pour tous les indices $i$, $j$, $M_{i,j}$ représente le poids de l'arc d'origine $i$ et d'extrémité $j$. Lorsque les sommets ne sont pas reliés, le poids de l'arc vaut l'infini. On définit la variable `inf=1e10` qui représente un poids infini.

**b)** Écrire une fonction récursive `Floyd1` qui admet comme arguments d'entrée la matrice d'adjacence `M`, le sommet de départ `i`, le sommet d'arrivée `j` et l'entier `k`. Cette fonction retourne $d_k(i,j)$ avec la programmation dynamique.

Écrire le programme principal permettant d'afficher la plus petite distance parcourue entre le sommet de départ 0 et le sommet d'arrivée 1 en utilisant la fonction `Floyd1`.

L'algorithme utilise-t-il la méthode « diviser pour régner » ?

**c)** Pour éviter de calculer plusieurs fois $d_k(i,j)$, on définit une liste `DIST` telle que $\text{DIST}[i][j][k] = d_k(i,j)$. L'indice $k$ varie entre 0 et $n$ inclus. Lorsque $\text{DIST}[i][j][k]$ n'a pas été calculé, $\text{DIST}[i][j][k] = -\infty$. Écrire une fonction récursive `Floyd2` qui admet comme arguments d'entrée la matrice d'adjacence `M`, le sommet de départ `i`, le sommet d'arrivée `j`, l'entier `k` et la liste `DIST` servant à stocker les résultats intermédiaires. Cette fonction retourne $d_k(i,j)$ avec la programmation dynamique.

Utilise-t-on la technique « *Top Down* » ou « *Bottom Up* » ?

**d)** Écrire une fonction itérative `Floyd3` qui admet comme arguments d'entrée la matrice d'adjacence `M`, le sommet `départ` et le sommet `arrivée`. Cette fonction retourne la distance d'un plus court chemin entre `départ` et `arrivée` en utilisant l'algorithme de Floyd-Warshall avec la programmation dynamique.

Utilise-t-on la technique « *Top Down* » ou « *Bottom Up* » ?

**e)** On souhaite afficher le chemin suivi en utilisant la liste `DIST`. On définit la liste `PRECEDENT` :

- Toutes les valeurs de `PRECEDENT` sont initialisées à $-1$.
- Si $i \neq j$ et si $-\infty < \text{DIST}[i][j][k] < \infty$, alors $\text{PRECEDENT}[i][j][k]$ est le sommet précédent $j$ sur un chemin optimal de $i$ à $j$ qui emprunte des sommets intermédiaires d'indice strictement inférieur à $k$.
- $\text{PRECEDENT}[i][j][0] = i$ si $i \neq j$ et $\text{M}[i,j] < \infty$.
- La valeur de $\text{PRECEDENT}[i][j][k]$ lorsque $i = j$ ou $\text{DIST}[i][j][k] = \infty$ ou $\text{DIST}[i][j][k] = -\infty$ n'a aucune importance. On peut la prendre égale à $-1$.
- $\text{PRECEDENT}[i][j][k] = \text{PRECEDENT}[i][j][k-1]$ si $d_k(i,j) = d_{k-1}(i,j)$.
- $\text{PRECEDENT}[i][j][k] = \text{PRECEDENT}[k-1][j][k-1]$ si
  $d_k(i,j) = d_{k-1}(i,k-1) + d_{k-1}(k-1,j)$ et $d_k(i,j) \neq \infty$ et $\text{PRECEDENT}[k-1][j][k-1] \neq -1$

Écrire le programme principal permettant d'afficher le chemin suivi entre les sommets `départ` et `arrivée`.

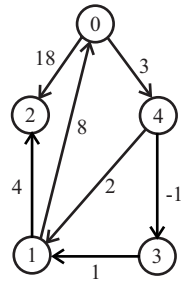
## 13.5

**Algorithme de Bellman-Ford – Programmation dynamique (sauf TSI et TPC)**

On considère le graphe orienté $G = (S, A)$ constitué de $n$ sommets. Les arêtes sont orientées et le poids des arcs peut être négatif. On définit la matrice d'adjacence $(M_{i,j})_{0 \leq i,j \leq n-1}$ du graphe. L'algorithme de Bellman-Ford définit $d_k(i,j)$ la distance minimale d'un chemin du sommet $i$ au sommet $j$ empruntant au plus $k$ arcs.

- Si $k = n-1$, alors $d_{n-1}(i,j)$ est la plus courte distance entre $i$ et $j$. Si le graphe n'a pas de cycle de poids négatif, il est inutile d'emprunter plus de $n-1$ arcs.
- Un chemin qui emprunte au plus zéro arc ne peut mener que de $i$ à $i$, donc $\begin{cases} d_0(i,j) = 0 & \text{si } i = j \\ d_0(i,j) = \infty & \text{si } i \neq j \end{cases}$.
- On considère un chemin optimal de $i$ à $j$ qui emprunte au plus $k$ arcs. On a deux possibilités :
  - soit ce chemin optimal de $i$ à $j$ emprunte au plus $k-1$ arcs ;
  - soit ce chemin se compose d'un chemin optimal de $i$ à $x$ en empruntant au plus $k-1$ arcs et d'un arc entre $x$ et $j$.

$$d_k(i,j) = \min_{0 \leq x < n} \left( d_{k-1}(i,j), d_{k-1}(i,x) + M[x,j] \right)$$




On utilise les listes de listes pour représenter les matrices dans Python.

`L.reverse()` permet d'inverser les éléments de la liste `L`.

**a)** Construire la matrice d'adjacence $\left(M_{i,j}\right)_{0\leq i,j\leq n-1}$ du graphe *G*, définie par : pour tous les indices *i, j*, $M_{i,j}$ représente le poids de l'arc d'origine *i* et d'extrémité *j*. Lorsque les sommets ne sont pas reliés, le poids de l'arc vaut l'infini. On définit la variable `inf=1e10` qui représente un poids infini.

**b)** Écrire une fonction récursive `Bellman_rec` qui admet comme arguments d'entrée la matrice d'adjacence `M`, le sommet de départ `i`, le sommet d'arrivée `j` et l'entier `k`. Cette fonction retourne $d_k(i,j)$ avec la programmation dynamique. On suppose que le graphe n'a pas de cycle de poids négatif.

Écrire le programme principal permettant d'afficher la plus petite distance entre le sommet de départ 0 et le sommet d'arrivée 1 en utilisant la fonction `Bellman_rec`.

L'algorithme utilise-t-il la méthode « diviser pour régner » ?

**c)** Pour éviter de calculer plusieurs fois $d_k(i,j)$, on définit une liste `DIST` telle que $\text{DIST}[i][j][k]=d_k(i,j)$. L'indice *k* varie entre 0 et *n* exclu. Lorsque $\text{DIST}[i][j][k]$ n'a pas été calculé, $\text{DIST}[i][j][k]=-\infty$. Écrire une fonction récursive `Bellman_rec2` qui admet comme arguments d'entrée la matrice d'adjacence `M`, le sommet de départ `i`, le sommet d'arrivée `j`, l'entier `k` et la liste `DIST` servant à stocker les résultats intermédiaires. Cette fonction retourne $d_k(i,j)$ avec la programmation dynamique. On suppose que le graphe n'a pas de cycle de poids négatif.

Utilise-t-on la technique « *Top Down* » ou « *Bottom Up* » ?

**d)** Écrire une fonction itérative `Bellman_iterative` qui admet comme arguments d'entrée la matrice d'adjacence `M`, le sommet `départ`, le sommet `arrivée` et la liste `DIST`. Cette fonction retourne la distance d'un plus court chemin entre `départ` et `arrivée` en utilisant l'algorithme de Bellman-Ford avec la programmation dynamique. On suppose que le graphe n'a pas de cycle de poids négatif. Utilise-t-on la technique « *Top Down* » ou « *Bottom Up* » ?

**e)** On définit une liste `PRECEDENT` : $\text{PRECEDENT}[i][j][k]$ est le sommet précédent *j* sur un chemin optimal de *i* à *j* qui emprunte au plus *k* arcs. Écrire une fonction itérative `Bellman_iterative2` qui admet comme arguments d'entrée la matrice d'adjacence `M`, le sommet `départ`, le sommet `arrivée`, la liste `DIST` et la liste `PRECEDENT`. On suppose que le graphe n'a pas de cycle de poids négatif. Écrire le programme principal permettant d'afficher le chemin suivi entre les sommets `départ` et `arrivée`.

**f)** On définit une liste `PRECEDENT` : $\text{PRECEDENT}[i][j][k]$ est le sommet précédent *j* sur un chemin optimal de *i* à *j* qui emprunte au plus *k* arcs. Écrire une fonction récursive `Bellman_rec3` qui admet comme arguments d'entrée la matrice d'adjacence `M`, le sommet de départ `i`, le sommet d'arrivée `j`, l'entier `k`, la liste `DIST` et la liste `PRECEDENT`. On suppose que le graphe n'a pas de cycle de poids négatif. Écrire le programme principal permettant d'afficher le chemin suivi entre les sommets `départ` et `arrivée`.

**g)** Comparer les algorithmes de Bellman-Ford et Dijkstra (voir exercice 12.1 « Algorithme de Dijkstra »). Utilise-t-on la méthode gloutonne ou la programmation dynamique ?

**h)** Si les éléments $d_k(i,j)$ changent entre l'étape $k = n–1$ et l'étape $k = n$, alors le graphe possède un cycle de poids négatif. Modifier le programme suivant pour détecter si le graphe possède un cycle de poids négatif.

```
inf=1e10      # variable représentant l'infini

def Bellman_iterative3 (M, départ, arrivée, DIST, PRECEDENT):
    n=len(M)                    # nombre de lignes de M
    for k in range         ##### A COMPLETER #####
        for i in range(n):      # i varie entre 0 inclus et n exclu
            for j in range(n):  # j varie entre 0 inclus et n exclu
                if k==0:
                    if j==i:
                        DIST[i][i][k]=0
                    else:
                        DIST[i][j][k]=inf
                else:
                    L=[DIST[i][j][k-1]]
                    L2=[-1]    # on ne connaît pas le sommet précédent
                    for x in range(n):
                        L.append(DIST[i][x][k-1]+M[x][j])
                        L2.append(x)      # le sommet précédent j est x
                    # recherche du minimum et de l'indice
                    mini=L[0]
                    indice=0
                    for j1 in range(1,len(L)):
                        if L[j1]<mini:
                            mini=L[j1]     # valeur du minimum
                            indice=L2[j1]  # sommet précédent j
                    if indice >0:
                        PRECEDENT[i][j][k]=indice
                    DIST[i][j][k]=mini
                    if DIST[i][j][k]==DIST[i][j][k-1]\
                      and DIST[i][j][k-1]>-inf and DIST[i][j][k-1]<inf:
                        PRECEDENT[i][j][k]=PRECEDENT[i][j][k-1]
    return DIST[départ][arrivée][n-1]

DIST=                      ##### A COMPLETER #####
PRECEDENT=                 ##### A COMPLETER #####
départ=0
arrivée=0
distance7=Bellman_iterative3(M, départ, arrivée, DIST, PRECEDENT)

# recherche si DIST est modifié entre l'étape k=n-1 et l'étape k=n

##### A COMPLETER #####
```

```
if rep==False:
    print("Le graphe ne possède pas de cycle de poids négatif.")
else:
    print("Le graphe possède au moins un cycle de poids négatif.")
```

# Du mal à démarrer ?

## 13.1

**a)** `n=0` et `n==1` sont des conditions d'arrêt.

**c)** Tester si `n` est dans `dico`. C'est une condition d'arrêt. Effectuer deux appels récursifs pour `n-1` et `n-2`.

**d)** Les tests `n=0` et `n==1` sont des conditions d'arrêt. Écrire une boucle `for` en commençant à 2.

## 13.2

a) Définir `Ltot=0` et `n` la longueur de la barre. Écrire une boucle `while` en partant de $i = 0$. Tester si la nouvelle longueur totale est inférieure ou égale à la longueur de la barre : `Ltot+S[i][1]<=n`.

**b)** `n<=1` est une condition d'arrêt. Écrire une boucle `for` en faisant varier `i` entre 1 inclus et `n` exclu. Effectuer deux appels récursifs pour `i` et `n-i`.

**c)** `L[j]>=0` est une condition d'arrêt. Stocker dans `L` le prix d'une barre de longueur `j`.

**d)** Écrire deux boucles `for` imbriquées : `j` varie entre 0 inclus et `n+1` exclu, `i` varie entre 1 inclus et `j` exclu.

**e)** Ajouter la ligne `T[j][indice_decoupe]=1`.

**f)** Définir une liste `ABSCISSES`. Écrire une boucle `for` : `i` varie entre `n` inclus et 0 exclu.

## 13.3

**a)** Écrire deux boucles `for` imbriquées : `i` varie entre 1 inclus et `n+1` exclu, `j` varie entre 1 inclus et `m+1` exclu.

**b)** `n==0` ou `m==0` est une condition d'arrêt. `n>m` est une condition d'arrêt. Effectuer deux appels récursifs pour (`n`, `m-1`) et (`n-1`, `m-1`).

**c)** `A[n, m]<inf` est une condition d'arrêt. `n==0` ou `m==0` est une condition d'arrêt. Stocker le résultat dans `A[n, m]`.

**d)** Écrire deux boucles `for` imbriquées : `i` varie entre 1 inclus et `n+1` exclu, `j` varie entre 0 inclus et `m+1` exclu.

## 13.4

**b)** `k<0` est une condition d'arrêt. Effectuer trois appels récursifs : (`i`, `j`, `k-1`), (`i`, `k-1`, `k-1`) et (`k-1`, `j`, `k-1`).

**c)** `k==0` est une condition d'arrêt. `DIST[i][j][k]!=-inf` est une condition d'arrêt.

**d)** Écrire trois boucles `for` imbriquées : `k` varie entre 0 inclus et `n+1` exclu, `i` varie entre 0 inclus et `n` exclu, `j` varie entre 0 inclus et `n` exclu.

**e)** Définir une liste `PRECEDENT`. Écrire deux boucles `for` imbriquées : `i` varie entre `n` inclus et 0 exclu.

### 13.5

b) k==0 est une condition d'arrêt. Appeler récursivement `Bellman_rec(M, i, j, k-1)`.

**c)** `k==0` et `DIST[i, j, k]!=-inf` sont des conditions d'arrêt.

**d)** Écrire trois boucles `for` imbriquées.

**e)** On obtient le chemin suivi en utilisant `PRECEDENT[départ, i, n-1]` puisqu'on a parcouru au plus $n-1$ arcs pour aller de `départ` à `arrivée`.

**f)** `k==0` et `DIST[i, j, k]!=-inf` sont des conditions d'arrêt. Appeler récursivement `Bellman_rec3(M, i, j, k-1, DIST, PRECEDENT)`.

# Corrigés des exercices

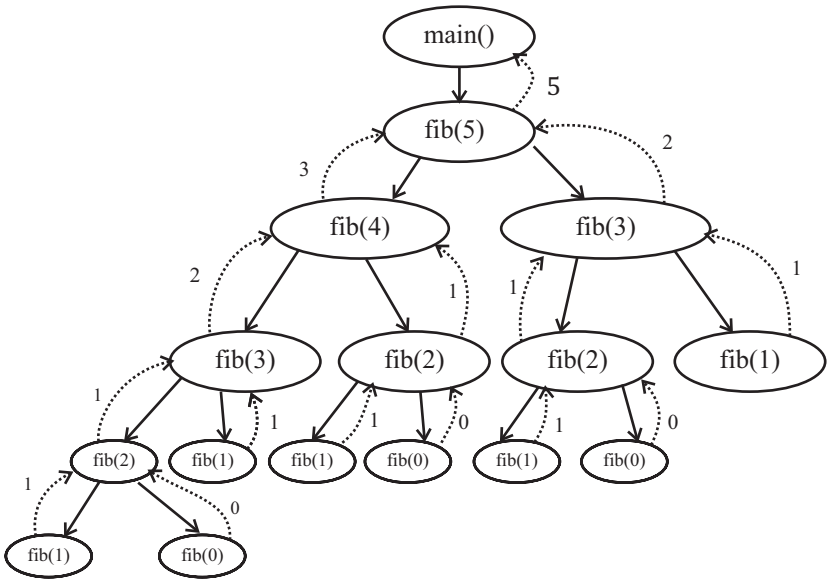
### 13.1

**a)**

```
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.

**b)** 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 $F_n$. On dit que l'on a un chevauchement de sous-problèmes.

**c) Technique récursive « *Top Down* »**

La technique de mémoïsation consiste à stocker les valeurs de $F_n$ dans une liste au fur et à mesure qu'elles sont calculées. On utilise un dictionnaire contenant les termes de la suite déjà calculés. Pour chaque élément du dictionnaire `dico`, on précise la clé et la valeur associée. La clé est l'entier `n` et la valeur est `F[n]`.

```
def fibo2(n, dico):
    # la fonction renvoie le nombre de Fibonacci F[n] pour l'entier n
    if n in dico:               # teste si n est dans dico
        return dico[n]          # condition d'arrêt – retourne F[n]
    else:
        if n==0:
            dico[n]=0           # calcul de F[0]
            return 0            # condition d'arrêt
        elif n==1:
            dico[n]=1           # calcul de F[1]
            return 1            # condition d'arrêt
        else:
            a=fibo2(n-1, dico)  # appel récursif
            if n-1 not in dico: # teste si la clé n-1 est dans dico
                dico[n-1]=a     # stockage de F[n-1] dans dico
            b=fibo2(n-2, dico)  # appel récursif
            if n-2 not in dico: # teste si la clé n-2 est dans dico
                dico[n-2]=b     # stockage de F[n-2] dans dico
            return (a+b)

dico={}
n=15
print('fibo2 =', fibo2(n, dico))
```

**Remarque**

On peut utiliser également une liste `L` contenant les termes de la suite déjà calculés :

- `L[n][0]` représente la valeur `Fn` ;
- `L[n][1]` vaut `False` si `Fn` n'a pas encore été calculé et vaut `True` dès que `Fn` est calculé.

```
def fibo2_liste(n):
    # la fonction renvoie le nombre de Fibonacci Fn pour l'entier n
    if L[n][1]==True:           # teste si F[n] est déjà calculé
        return L[n][0]          # condition d'arrêt – retourne F[n]
    else:
        if n==0:
            L[n]=0, True        # stockage de F[0]
            return 0            # condition d'arrêt
```

```
        elif n==1:
            L[n]=1, True        # stockage de F[1]
            return 1            # condition d'arrêt
        else:
            a=fibo2_liste(n-1)  # appel récursif
            L[n-1]=a, True      # stockage de F[n-1]
            b=fibo2_liste(n-2)  # appel récursif
            L[n-2]=b, True      # stockage de F[n-2]
            return (a+b)
```

**d) Technique itérative « *Bottom Up* »**

La fonction `fibo3` est une fonction itérative qui prend en argument un entier naturel $n$ et renvoie le nombre de Fibonacci $F_n$ en utilisant la technique « *Bottom Up* ».

La relation de récurrence s'écrit pour $n \geq 2$ : $F_n = F_{n-1} + F_{n-2}$.

```
def fibo3(n):
    # la fonction renvoie le nombre de Fibonacci Fn pour l'entier n
    # on remplit dans une liste les valeurs de F[n]
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        F=[0, 1]                 # initialisation de la liste F
        for i in range(2, n+1):  # i varie entre 2 inclus et n+1 exclu
            F.append(F[i-1]+F[i-2])
        return F[i]
```

L'algorithme « *Bottom Up* » résout tous les sous-problèmes de taille inférieure alors que l'algorithme « *Top Down* » ne résout que les sous-problèmes de taille inférieure dont il a besoin.

## 13.2

**a)** 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 longueur 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 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.

**b)** La méthode « diviser pour régner » peut se décomposer en trois étapes :

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

L'algorithme est de type « diviser pour régner » puisqu'on décompose le problème (calcul de $M[n]$) en plusieurs sous-problèmes : calcul de $M[1]+M[n-1]$, $M[2]+M[n-2]$, …, $M[n-1]+M[1]$. On combine les différents sous-problèmes pour résoudre le problème de départ. L'algorithme est bien acyclique car $i > 0$ et $n - i < n$. On traite bien des sous-problèmes que l'on peut traiter de façon optimale.

**Remarques**

- Si on avait $i = 0$, le calcul de $M[n]$ nécessiterait de calculer $M[0]+M[n]$. On aurait alors un débordement de la pile des appels récursifs !
- On remarque qu'il y a une symétrie dans les calculs des sous-problèmes. On pourrait réduire le nombre de sous-problèmes pour le calcul de $M[n]$ : `for i in range(1,(n+1)//2):` au lieu de `for i in range(1,n):`.

Le principal inconvénient est que l'on calcule plusieurs fois $M[i]$. On dit que l'on a un chevauchement de sous-problèmes. On va utiliser dans la question suivante la technique de mémoïsation qui consiste à stocker dans une liste les valeurs déjà calculées.

```
def decoupe2(P, n):
    # la fonction retourne le prix optimal que l'on peut obtenir d'une barre
    # de longueur n
    if n<=1:
```

```
        return P[n]                    # condition d'arrêt
    else:
        maxi=0 #initialisation de maxi
        for i in range(1,(n+1)//2):   # en exploitant la symétrie
            somme=decoupe2(P, i)+decoupe2(P, n-i)   # appels récursifs
            if somme>maxi:
                maxi=somme
        if P[n]>maxi:
            maxi=P[n]
        return maxi

P=[0, 3, 5, 14, 16, 22]               # 6 éléments pour une barre de longueur 5
n=len(P)-1                            # n = longueur de la barre
prix=decoupe2(P, n)
print("Prix découpe2 :", prix)
```

**c)** Pour éviter de calculer plusieurs fois la même valeur totale, on garde en mémoire le résultat dans la liste $L$ : $L[i]$ contient le prix optimal pour une barre de longueur $i$. La liste $L$ doit contenir $n$+1 valeurs. On initialise tous les éléments à $-1$, c'est-à-dire qu'ils n'ont pas encore été traités.

Pour une barre de longueur $j$, on calcule récursivement $M(j)=\max\left(p(j),\max_{0<i<j}\left(M(i)+M(j-i)\right)\right)$ :

- Si $L[j]\geq 0$, on a déjà calculé le prix optimal d'une barre de longueur $j$. Il suffit de retourner cette valeur. Le sous-problème est déjà résolu. C'est une condition d'arrêt de la fonction récursive.
- Si $j\leq 1$, on ne peut pas découper la barre. Le prix de cette barre vaut $P[j]$. Le sous-problème est déjà résolu. C'est une condition d'arrêt de la fonction récursive.
- Si les conditions précédentes ne sont pas réunies, on calcule récursivement $M(i)$ et $M(j-i)$ en appelant les fonctions `decoupe3(P, i, L)` et `decoupe3(P, j-i, L)`. Il faut ensuite calculer $\max\left(p(j),\max_{0<i<j}\left(M(i)+M(j-i)\right)\right)$.

```
def decoupe3(P, j, L):
    # la fonction retourne le prix optimal que l'on peut obtenir d'une barre
    # de longueur j. Stockage des résultats intermédiaires dans L
    # L[i] = prix optimal d'une barre de longueur i
    if L[j]>=0:                       # condition d'arrêt
        return L[j]
    else:
        if j<=1:
            L[j]=P[j]
            return L[j]               # condition d'arrêt
        else:
            maxi=0                    # initialisation de maxi
            for i in range(1, j):     # i varie entre 1 inclus et j exclu
                somme=decoupe3(P, i, L)+decoupe3(P, j-i, L)
                if somme>maxi:
                    maxi=somme
```

```
            if P[j]>maxi:
                maxi=P[j]
            L[j]=maxi      # stockage dans L du prix d'une barre de longueur j
            return maxi

P=[0, 3, 5, 14, 16, 22]    # 6 éléments pour une barre de longueur 5
n=len(P)-1                 # n = longueur de la barre
L=[-1 for i in range(n+1)] # initialisation des éléments à -1
prix=decoupe3(P, n, L)
print("Prix découpe3 :", prix)
```

La fonction `decoupe3` utilise la programmation dynamique avec la technique « *Top Down* » (de mémoïsation) et permet d'obtenir une solution optimale sans résoudre plusieurs fois le même sous-problème.

**d)**

```
def decoupe4(P, n, L):
    # la fonction retourne le prix optimal que l'on peut obtenir d'une barre
    # de longueur n. Stockage des résultats intermédiaires dans L
    # L[i] = prix optimal d'une barre de longueur i
    for j in range(n+1):     # j varie entre 0 inclus et n+1 exclu
        maxi=0               # initialisation de maxi
        for i in range(1,j): # i varie entre 1 inclus et j exclu
            somme=L[i]+L[j-i]
            if somme>maxi:
                maxi=somme
        if P[j]>maxi:
            maxi=P[j]
        L[j]=maxi            # stockage dans L du prix d'une barre de longueur j
    return L[n]
```

La fonction `decoupe4` utilise la programmation dynamique avec la technique « *Bottom Up* » : on utilise la même formule de récurrence que dans `decoupe2` et `decoupe3` mais on part d'une barre de longueur nulle au lieu de partir d'une barre de longueur *n* (technique « *Top Down* »).

- Dans la technique « *Top Down* », on ne traite que les sous-problèmes nécessaires. On n'a pas besoin de remplir entièrement la liste `L` pour obtenir la solution optimale au problème.
- Dans la technique « *Bottom Up* », on traite tous les sous-problèmes. Il faut d'abord remplir entièrement la liste `L` avant de retourner la solution optimale au problème (`L[n]`).

```
P=[0, 3, 5, 14, 16, 22]      # 6 éléments pour une barre de longueur 5
n=len(P)-1                   # n = longueur de la barre
L=[-1 for i in range(n+1)]   # initialisation des éléments à -1
prix=decoupe4(P, n, L)
print("Prix découpe4 :",prix)
```

**e)** En dessous des lignes `L[j]=P[j]` et `L[j]=maxi`, on ajoute `T[j, j]=1` pour préciser qu'on a effectué une découpe à l'abscisse *j* pour une barre de longueur *j*.

En dessous de la ligne `L[j]=maxi` après `else`, on ajoute `T[j, indice_decoupe]=1` pour préciser qu'on a effectué une découpe à l'abscisse `indice_decoupe` pour une barre de longueur *j*.

```
def decoupe5(P, j, L):
    # la fonction retourne le prix optimal que l'on peut obtenir d'une barre
    # de longueur j. Stockage des résultats intermédiaires dans L
    # L[i] = prix optimal d'une barre de longueur i
    if L[j]>=0:                     # condition d'arrêt
        return L[j]
    else:
        if j<=1:
            L[j]=P[j]
            T[j][j]=1
            return L[j]             # condition d'arrêt
        else:
            maxi=0
            indice_decoupe=0
            for i in range(1,j):    # i varie entre 1 inclus et j exclu
                somme=decoupe5(P, i, L)+decoupe5(P, j-i, L)
                if somme>maxi:
                    indice_decoupe=i
                    maxi=somme
            if P[j]>maxi:
                maxi=P[j]
                L[j]=maxi
                T[j][j]=1
            else:                   # découpe à l'abscisse indice_decoupe
                L[j]=maxi
                T[j][indice_decoupe]=1
            return maxi

P=[0, 3, 5, 14, 16, 22]             # 6 éléments pour une barre de longueur 5
n=len(P)-1                          # n = longueur de la barre
L=[-1 for i in range(n+1)]           # initialisation des éléments à -1
T=[[0 for j in range(n+1)] for i in range(n+1)]
prix=decoupe5(P, n, L)
print("Prix découpe5 :",prix)

ABSCISSES=[]                        # création de la liste des abscisses
m=n
total=0
while m>0:
    for i in range(len(P)-1, 0, -1): # i désigne le ième objet
        if T[m][i]==1:
            ABSCISSES.append(i)     # on a découpé à l'abscisse i
            total+=P[i]
            m=m-i                   # retranche i à la longueur de la barre
print('Abscisses 5 :', ABSCISSES, total)
```

**f)** En dessous de la ligne `L[j]=maxi`, on ajoute `T[j][j]=1` pour préciser qu'on a effectué une découpe à l'abscisse *j* pour une barre de longueur *j*.

En dessous de la ligne `L[j]=maxi` après `else`, on ajoute `T[j][indice_decoupe]=1` pour préciser qu'on a effectué une découpe à l'abscisse `indice_decoupe` pour une barre de longueur *j*.

```
def decoupe6(P, n, L):
    # la fonction retourne le prix optimal que l'on peut obtenir d'une barre
    # de longueur n. Stockage des résultats intermédiaires dans L
    # L[i] = prix optimal d'une barre de longueur i
    for j in range(n+1):     # j varie entre 0 inclus et n+1 exclu
        maxi=0               # initialisation de maxi
        indice_decoupe=0
        for i in range(1,j): # i varie entre 1 inclus et j exclu
            somme=L[i]+L[j-i]
            if somme>maxi:
                indice_decoupe=i
                maxi=somme
        if P[j]>maxi:
            maxi=P[j]
            L[j]=maxi
            T[j][j]=1
        else:                # découpe à l'abscisse indice_decoupe
            L[j]=maxi        # stockage dans L du prix d'une barre de longueur j
            T[j][indice_decoupe]=1
    return L[n]

P=[0, 3, 5, 14, 16, 22]      # 6 éléments pour une barre de longueur 5
n=len(P)-1                   # n = longueur de la barre
L=[-1 for i in range(n+1)]    # initialisation des éléments à -1
T=[[0 for j in range(n+1)] for i in range(n+1)]
prix=decoupe6(P, n, L)
print("Prix découpe6 :",prix)

ABSCISSES=[]                          # création de la liste des abscisses
m=n
total=0
while m>0:
    for i in range(len(P)-1, 0, -1): # i désigne le ième objet
        if T[m][i]==1:
            ABSCISSES.append(i)      # on a découpé à l'abscisse i
            total+=P[i]
            m=m-i                    # retranche i à la longueur de la barre
print('Abscisses 6 :', ABSCISSES,total)
```

## 13.3

a)

```
inf=1e10                        # variable représentant l'infini
def optim1(n, m):               # méthode gloutonne
    # retourne total (fonction d'évaluation) et liste attribution F
    # correspondant aux n premiers skieurs et m premières paires de skis
    total=0
    T=[0 for i in range(m+1)]  # 1 si paire de skis déjà attribué, 0 sinon
    F=[0 for i in range(m+1)]
    for i in range(1,n+1):      # i varie entre 1 inclus et n inclus
        mini=inf
        indice=0
        for j in range(1, m+1):
            if abs(S[j]-L[i])<mini and T[j]==0:  # paire de skis j non attribué
                mini=abs(S[j]-L[i])
                indice=j
        total=total+abs(S[indice]-L[i])
        F[i]=indice
        T[indice]=1           # on a attribué la paire de skis de numéro indice
            # on ne revient plus sur ce choix -> méthode gloutonne
    return total, F
```

Cette méthode est appelée méthode gloutonne car elle consiste à faire le meilleur choix sur le moment, c'est-à-dire attribuer la paire de skis qui a la longueur la plus proche de la taille du skieur.

La fonction `optim1` retourne la liste `F` = [0, 2, 3, 4, 5].

Skieur 140 → paire de skis de longueur 148

Skieur 150 → paire de skis de longueur 153

Skieur 150 → paire de skis de longueur 160

Skieur 162 → paire de skis de longueur 180

La fonction d'évaluation vaut $|148-140|+|153-150|+|160-150|+|180-162|=39$.

Le problème est que l'on a fait un mauvais choix pour le premier ski. On ne revient plus sur ce choix. On pourra résoudre ce problème avec la programmation dynamique.

**b)** La méthode « diviser pour régner » peut se décomposer en trois étapes :

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

L'algorithme est de type « diviser pour régner » puisqu'on décompose le problème (calcul de `A[n][m]`) en plusieurs sous-problèmes : calcul de `A[n][m-1]`, `A[n-1][m-1]`. On combine les différents sous-problèmes pour résoudre le problème de départ. L'algorithme est bien acyclique car `m-1` < `m` et `n-1` < `n`. On traite bien des sous-problèmes que l'on peut traiter de façon optimale.

Le principal inconvénient est que l'on calcule plusieurs fois `A[i][j]`. On dit que l'on a un chevauchement de sous-problèmes. On utilisera dans la question suivante la technique de mémoïsation qui consiste à stocker dans une liste les valeurs déjà calculées.

```
def optim2(n, m):
    # retourne la fonction d'évaluation optimale pour le problème restreint
    # aux n premiers skieurs et m premières paires de skis
    if n==0 or m==0:       # condition d'arrêt avec 0 skieur ou 0 paire de skis
        return 0
    elif n>m:              # trop de skieurs par rapport aux paires de skis
        return inf         # condition d'arrêt
    else:
        a=optim2(n, m-1)   # appel récursif
        b=optim2(n-1, m-1)+abs(S[m]-L[n])   # appel récursif
        if a<b and m>=2:   # il faut avoir au moins deux paires
                           # de skis pour tester m-1
            mini=a
        else:
            mini=b
        return mini
```

**c)** Pour éviter de calculer plusieurs fois la même fonction d'évaluation, on garde en mémoire le résultat dans la liste `A` : `A[i][j]` contient la fonction d'évaluation optimale du problème restreint aux `i` premiers skieurs et `j` premières paires de skis. On initialise à l'infini (variable `inf`) tous les éléments de `A` qui n'ont pas encore été traités.

Pour calculer `A[n][m]`, on calcule récursivement $\min\left(A[n][m-1], \left(A[n-1][m-1]+\left|S[m]-L[n]\right|\right)\right)$ :

- Si `A[n][m]` < `inf`, on a déjà calculé la fonction d'évaluation optimale. Il suffit de retourner cette valeur. Le sous-problème est déjà résolu. C'est une condition d'arrêt de la fonction récursive.
- Si $n = 0$ ou $m = 0$. On considère 0 skieur ou 0 paire de skis. La fonction d'évaluation vaut : `A[n][m]=0`. Le sous-problème est déjà résolu. C'est une condition d'arrêt de la fonction récursive.
- Si $n > m$, il y a trop de skieurs par rapport au nombre de paires de skis. La fonction d'évaluation vaut : `A[n][m]=inf`. Le sous-problème est déjà résolu. C'est une condition d'arrêt de la fonction récursive.
- Si les conditions précédentes ne sont pas réunies, on calcule récursivement `A[n][m-1]` et `A[n-1][m-1]`. On a alors $\left(A[n][m] = \min\left(A[n][m-1],\right)\left(A[n-1][m-1]+\left|S[m]-L[n]\right|\right)\right)$.

```
def optim3(A, n, m):          # programmation dynamique top down
    # retourne la fonction d'évaluation optimale pour le problème restreint
    # aux n premiers skieurs et m premières paires de skis
    if A[n][m]<inf:           # condition d'arrêt
        return A[n][m]
    else:
        if n==0 or m==0:      # condition d'arrêt avec 0 skieur ou 0 ski
            A[n][m]=0
            return 0
        elif n>m:   # trop de skieurs par rapport au nombre de paires de skis
            return inf        # condition d'arrêt - valeur infinie
        else:
            a=optim3(A, n, m-1)
            b=optim3(A, n-1, m-1)+abs(S[m]-L[n])
            if a<b and m>=2: # il faut avoir au moins deux paires
                             # de skis pour tester m-1
```

```
                mini=a
            else:
                mini=b
            A[n][m]=mini
            return mini

A=[[inf for j in range(m+1)] for i in range(n+1)]
                            # A[i][j]   i : skieur et j : paire de skis
resultat3=optim3(A, n, m)
print('résultat 3 =',resultat3)
```

La fonction `optim3` utilise la programmation dynamique avec la technique « *Top Down* » de mémoïsation et permet d'obtenir une solution optimale sans résoudre plusieurs fois le même sous-problème.

**d)** La fonction `optim4` utilise la programmation dynamique avec la technique « *Bottom Up* » : on utilise la même formule de récurrence que dans `optim2` et `optim3` mais on part de `i=0` et `j=0` au lieu de partir de `n` et `m` (technique « *Top Down* »).

- Dans la technique « *Top Down* », on ne traite que les sous-problèmes nécessaires. On n'a pas besoin de remplir entièrement la liste `A` pour obtenir la solution optimale au problème.
- Dans la technique « *Bottom Up* », on traite tous les sous-problèmes. Il faut d'abord remplir entièrement la liste `A` avant de retourner la solution optimale au problème `A[n, m]`.

```
def optim4(A,n,m):
    # retourne la fonction d'évaluation optimale pour le problème restreint
    # aux n premiers skieurs et m premières paires de skis
    for i in range(n+1):          # programmation dynamique bottom up
        for j in range(m+1):
            if i<=m:
                A[i][0]=0
                A[0][j]=0

    for i in range(1, n+1):
        for j in range(m+1):
            if i<=j:
                a=optim3(A, i, j-1)
                b=optim3(A, i-1, j-1)+abs(S[j]-L[i])
                if a<b and j>=2: # il faut avoir au moins deux paires
                                 # de skis pour tester m-1
                    mini=a
                else:
                    mini=b
                A[i][j]=mini
    return A[n][m]

A=[[inf for j in range(m+1)] for i in range(n+1)]
                                 # A[i][j]   i : skieur et j : paire de skis
resultat4=optim4(A, n, m)
print('résultat 4 =',resultat4)
```

**e)** Il y a deux cas dans la récurrence :

- Si la paire de skis `m` n'est associée à aucun skieur : $A[n][m] = A[n][m-1]$.
- Si la paire de skis `m` est associée au skieur `n` : $A[n][m] = A[n-1][m-1] + |S[m] - L[n]|$. Dans ce cas, on peut affecter la liste `F[n]=m`.

On ajoute alors une ligne `T[n][m]=1` pour indiquer que l'on a attribué la $m^{\text{ième}}$ paire de skis au $n^{\text{ième}}$ skieur.

Après l'appel de la fonction récursive `optim5(A, T, n, m)`, on ajoute dans le programme principal deux boucles pour remplir la liste d'attribution `F` :

- On commence par le $n^{\text{ième}}$ skieur. On parcourt la liste `T[i][j]` avec $i = n$ et $j = m$. On commence par la paire de skis la plus grande et on fait décroître $j$ jusqu'à avoir une valeur non nulle de `T[i][j]`.
- Si on a trouvé une valeur non nulle de T[i][j], alors on peut alors attribuer la $j^{\text{ème}}$ paire de skis au $i^{\text{ème}}$ skieur et on décrémente j de 1.

On continue la boucle `for` avec le $(n-1)^{\text{ième}}$ skieur. On parcourt `T[i][j]` en partant de la valeur de $j$ définie obtenue juste avant.

```
def optim5(A, T, n, m):
    # arguments d'entrée : n et m entiers, A et T matrices
    if A[n][m]<inf:           # condition d'arrêt
        return A[n][m]
    else:
        if n==0 or m==0:      # condition d'arrêt avec 0 skieur
                              # ou 0 paire de skis
            A[n][m]=0
            return 0
        elif n>m:             # trop de skieurs pour le nombre de paires de skis
            return inf        # condition d'arrêt - valeur infinie
        else:
            a=optim5(A, T, n, m-1)
            b=optim5(A, T, n-1, m-1)+abs(S[m]-L[n])
            if a<b and m>=2:  # il faut avoir au moins deux paires
                              # de skis pour tester m-1
                mini=a
            else:
                mini=b
                T[n][m]=1
            A[n][m]=mini
            return mini

A=[[inf for j in range(m+1)] for i in range(n+1)]
                              # A[i][j]   i : skieur et j : paire de skis
T=[[0 for j in range(m+1)] for i in range(n+1)]
resultat5=optim5(A, T, n, m)
print('résultat 5 =',resultat5)

# création de la liste d'attribution F
F=[0 for i in range(n+1)]
```

```
j=m
for i in range(n, 0, -1):     # i varie entre n inclus et 0 exclu
    flag=False
    while flag==False:
         if T[i][j]!=0:
             flag=True
             F[i]=j
         j=j-1
print(F)
```

## 13.4

**a)** La matrice d'adjacence est : $M = \begin{pmatrix} 0 & \infty & 18 & \infty & 3 \\ 8 & 0 & 4 & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \\ \infty & 1 & \infty & 0 & \infty \\ \infty & 2 & \infty & -1 & 0 \end{pmatrix}$.

Comme le graphe est orienté, la matrice n'est pas nécessairement symétrique.

**b)** 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.

L'algorithme est de type « diviser pour régner » puisqu'on décompose le problème (calcul $d_k(i,j)$ en plusieurs sous-problèmes (calcul de $d_{k-1}(i,j), d_{k-1}(i,k-1), d_{k-1}(k-1,j)$ ). On combine les différents sous-problèmes pour résoudre le problème de départ.

Dans la programmation dynamique, on calcule toutes les solutions des sous-problèmes que l'on combine pour obtenir une solution optimale.

```
def Floyd1 (M, i, j, k):
    # la fonction retourne d_k(i,j) pour la matrice d'adjacence M
    if k==0:                    # condition d'arrêt
        return M[i][j]
    else:
        a=Floyd1(M, i, j, k-1)  # appel récursif
        b=Floyd1(M, i, k-1, k-1)+Floyd1(M, k-1, j, k-1)
        return (min(a,b))

inf=1e10                        # variable représentant l'infini

M= [[0, inf, 18, inf, 3],
   [8, 0, 4, inf, inf],
   [inf, inf, 0, inf, inf],
   [inf, 1, inf, 0, inf],
   [inf, 2, inf, -1, 0]]
n=len(M)                        # nombre de lignes de M
```

```
départ=0                          # sommet de départ
arrivée=1                         # sommet d'arrivée
distance1=Floyd1(M, départ, arrivée, n)
print("Distance parcourue Floyd1 :", distance1)
```

**c)** On rencontre deux techniques dans la programmation dynamique :

- Technique récursive « *Top Down* » de mémoïsation : on résout dans le sens des données de grande taille vers les données de petite taille. Lors d'un appel récursif, on regarde dans une liste intermédiaire si le sous-problème est déjà traité.
- Technique itérative « *Bottom Up* » : on résout dans le sens des données de petite taille vers les données de grande taille (c'est l'ordre inverse de « *Top Down* »). On stocke également les résultats obtenus dans une liste intermédiaire.

La fonction `Floyd2` utilise la programmation dynamique avec la technique « *Top Down* » (de mémoïsation) et permet d'obtenir une solution optimale sans résoudre plusieurs fois le même sous-problème.

Dans le programme principal, on part de la plus grande valeur de $k$ (ici $n$) : `distance2=Floyd2(M, départ, arrivée, n, DIST)`. On décompose le problème (calcul $d_k(i,j)$ en plusieurs sous-problèmes (calcul de $d_{k-1}(i,j), d_{k-1}(i,k-1), d_{k-1}(k-1,j)$ ).

Pour éviter de calculer plusieurs fois $d_k(i,j)$, on garde en mémoire le résultat dans la liste `DIST` : $\text{DIST}[i][j][k] = d_k(i,j)$. La liste doit contenir $(n)(n)(n+1)$ valeurs.

- $i$ et $j$ varient entre 0 inclus et $n$ exclu.
- $k$ varie entre 0 inclus et $n$ inclus.

Lorsque $\text{DIST}[i][j][k]$ n'a pas été calculé, $\text{DIST}[i][j][k] = -\infty$.

```
def Floyd2(M, i, j, k, DIST):
    # la fonction retourne d_k(i,j) pour la matrice d'adjacence M
    # la liste DIST est telle que DIST[i ][j ][k]=d_k(i,j)
    if k==0:                          # condition d'arrêt
        return M[i][j]
    elif DIST[i][j][k]!=-inf:         # condition d'arrêt
        return DIST[i][j][k]
    else:
        a=Floyd2(M, i, j, k-1, DIST)  # appel récursif
        b=Floyd2(M, i, k-1, k-1, DIST)+Floyd2(M, k-1, j, k-1, DIST)
        DIST[i][j][k]=min(a,b)
        return  DIST[i][j][k]

DIST=[[[-inf for k in range(n+1)] for j in range(n)] for i in range(n)]
          # toutes les distances valent -inf
départ=0                              # sommet de départ
arrivée=1                             # sommet d'arrivée
distance2=Floyd2(M, départ, arrivée, n, DIST)
print("Distance parcourue Floyd2 :", distance2)
```

**d)** L'algorithme `Floyd3` utilise la programmation dynamique avec la technique « *Bottom Up* » : on utilise la même formule de récurrence mais on part de la plus petite valeur de $k$ au lieu de partir de la plus grande valeur de $k$ ($k = n$, technique « *Top Down* »).

Il faut trois boucles `for` imbriquées pour faire varier $i$, $j$ et $k$.

- Dans la technique « *Top Down* », on ne traite que les sous-problèmes nécessaires. On n'a pas besoin de remplir entièrement la liste `DIST` pour obtenir la solution optimale au problème.
- Dans la technique « *Bottom Up* », on traite tous les sous-problèmes (trois boucles `for` imbriquées). Il faut d'abord remplir entièrement la liste `DIST` avant de retourner la solution optimale au problème.

```
def Floyd3(M, départ, arrivée, DIST):
    # la fonction retourne DIST[départ ][arrivée ][n]
    # n = nombre de sommets
    n=len(M)                    # nombre de lignes de M
    for k in range (n+1):       # k varie entre 0 inclus et n+1 exclu
        for i in range(n):      # i varie entre 0 inclus et n exclu
            for j in range(n):  # j varie entre 0 inclus et n exclu
                if k==0:
                    DIST[i][j][k]=M[i][j]
                else:
                    a=DIST[i][j][k-1]
                    b=DIST[i][k-1][k-1]+DIST[k-1][j][k-1]
                    DIST[i][j][k]=min(a,b)
    return DIST[départ][arrivée][n]

DIST=[[[-inf for k in range(n+1)] for j in range(n)] for i in range(n)]
          # toutes les distances valent -inf
départ=0                        # sommet de départ
arrivée=1                       # sommet d'arrivée
distance3=Floyd3(M, départ, arrivée, DIST)
print("Distance parcourue Floyd3 :", distance3)
```

**e)** On utilise l'algorithme `Floyd3` avec la technique « *Bottom Up* » pour obtenir la liste `DIST` entièrement remplie.

Pour $k = 0$, on considère deux boucles `for` imbriquées pour calculer $\text{PRECEDENT}[i, j, 0]$ :

- $\text{PRECEDENT}[i][j][0] = i$ si $i \neq j$ et $\text{M}[i, j] < \infty$.

On utilise trois boucles `for` imbriquées pour calculer $\text{PRECEDENT}[i][j][k]$ :

- $\text{PRECEDENT}[i][j][k] = \text{PRECEDENT}[i][j][k-1]$ si $d_k(i, j) = d_{k-1}(i, j)$.
- $\text{PRECEDENT}[i][j][k] = \text{PRECEDENT}[k-1][j][k-1]$ si

  $d_k(i, j) = d_{k-1}(i, k-1) + d_{k-1}(k-1, j)$ et $d_k(i, j) \neq \infty$ et $\text{PRECEDENT}[k-1][j][k-1] \neq -1$

```
PRECEDENT=[[[-1 for k in range(n+1)] for j in range(n)] for i in range(n)]
        # tous les éléments valent -1
for i in range(n):
    for j in range(n):
        if i!=j and M[i][j]<inf:
            PRECEDENT[i][j][0]=i
for k in range(1, n+1):
    for i in range(n):
        for j in range(n):
            if DIST[i][j][k]==DIST[i][j][k-1]:
```

```
                PRECEDENT[i][j][k]=PRECEDENT[i][j][k-1]
            if DIST[i][j][k]==DIST[i][k-1][k-1]+DIST[k-1][j][k-1] \
            and DIST[i][j][k]!=inf and PRECEDENT[k-1][j][k-1]!=-1:
                PRECEDENT[i][j][k]=PRECEDENT[k-1][j][k-1]


# recherche du chemin suivi entre les sommets départ et arrivée
indice_boucle=0
i=arrivée
chemin=[arrivée]   # initialisation de la liste des sommets parcourus
                   # en sens inverse
while (i!=départ) and indice_boucle<n:
    i=int(PRECEDENT[départ][i][n])
    chemin.append(i)
    indice_boucle=indice_boucle+1
# on obtient la liste des sommets en sens inverse
chemin.reverse()   # il faut inverser la liste chemin
print(chemin)
```

## 13.5

a) La matrice d'adjacence est : $M = \begin{pmatrix} 0 & \infty & 18 & \infty & 3 \\ 8 & 0 & 4 & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \\ \infty & 1 & \infty & 0 & \infty \\ \infty & 2 & \infty & -1 & 0 \end{pmatrix}$.

Comme le graphe est orienté, la matrice n'est pas nécessairement symétrique.

**b)** 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.

L'algorithme est de type « diviser pour régner » puisqu'on décompose le problème (calcul $d_k(i,j)$ en plusieurs sous-problèmes (calcul de $d_{k-1}(i,j), d_{k-1}(i,x)$). On combine les différents sous-problèmes pour résoudre le problème de départ.

Dans la programmation dynamique, on calcule toutes les solutions des sous-problèmes que l'on combine pour obtenir une solution optimale.

Pour chercher $\min\limits_{0\le x<n}\left(d_{k-1}(i,j), d_{k-1}(i,x)+M[x][j]\right)$, on crée une liste $L$ avec un élément $d_{k-1}(i,j)$. On ajoute les autres éléments $d_{k-1}(i,x)+M[x][j]$ avec une boucle `for x in range(n):`. Le minimum de la liste $L$ s'obtient par `min(L)`.

```
inf=1e10                    # variable représentant l'infini


def Bellman_rec  (M, i, j, k):
    # la fonction retourne d_k(i,j) pour la matrice d'adjacence M
    n=len(M)                # nombre de lignes de M
```

```
    if k==0:                # condition d'arrêt
        if j==i:
            return 0
        else:
            return inf
    else:
        L=[Bellman_rec(M, i, j, k-1)]                    # appel récursif
        for x in range(n):
            L.append(Bellman_rec(M, i, x, k-1)+M[x][j]) # appel récursif
        return min(L)

M= [[0, inf, 18, inf, 3],
   [8, 0, 4, inf, inf],
   [inf, inf, 0, inf, inf],
   [inf, 1, inf, 0, inf],
   [inf, 2, inf, -1, 0]]
n=len(M)                    # nombre de lignes de M
départ=0                    # sommet de départ
arrivée=1                   # sommet d'arrivée
distance2=Bellman_rec (M, départ, arrivée, n-1)
print("Distance parcourue Bellman_rec :", distance2)
```

On peut montrer que si le graphe n'a pas de cycle de poids négatif, il est inutile d'emprunter plus de $n-1$ arcs. On appelle donc la fonction récursive `Bellman_rec(M, i, j, k)` avec $k = n-1$.

**c)** On rencontre deux techniques dans la programmation dynamique :

- Technique récursive « *Top Down* » de mémoïsation : on résout dans le sens des données de grande taille vers les données de petite taille. Lors d'un appel récursif, on regarde dans une liste intermédiaire si le sous-problème est déjà traité.
- Technique itérative « *Bottom Up* » : on résout dans le sens des données de petite taille vers les données de grande taille (c'est l'ordre inverse de « *Top Down* »). On stocke également les résultats obtenus dans une liste intermédiaire.

La fonction `Bellman_rec2` utilise la programmation dynamique avec la technique « *Top Down* » (de mémoïsation) et permet d'obtenir une solution optimale sans résoudre plusieurs fois le même sous-problème.

Dans le programme principal, on part de la plus grande valeur de $k$ (ici $n-1$) : `distance3=Bellman_rec2(M, départ, arrivée, n-1, DIST)`. On décompose le problème (calcul $d_k(i,j)$ en plusieurs sous-problèmes (calcul de $d_{k-1}(i,j), d_{k-1}(i,x)$).

Pour éviter de calculer plusieurs fois $d_k(i,j)$, on garde en mémoire le résultat dans la liste `DIST` : $\text{DIST}[i][j][k] = d_k(i,j)$. La liste doit contenir $n^3$ valeurs.

- $i$ et $j$ varient entre 0 inclus et $n$ exclu.
- $k$ varie entre 0 inclus et $n$ exclu (d'après l'énoncé il est inutile d'emprunter plus de $n-1$ arcs).

Lorsque $\text{DIST}[i][j][k]$ n'a pas été calculé, $\text{DIST}[i][j][k] = -\infty$.

```
def Bellman_rec2(M, i, j, k, DIST):
    # la fonction retourne d_k(i,j) pour la matrice d'adjacence M
    # la liste DIST est telle que DIST[i][j][k]=d_k(i,j)
```

```
    n=len(M)                                    # nombre de lignes de M
    if k==0:                                    # condition d'arrêt
        if j==i:
            return 0
        else:
            return inf
    elif DIST[i][j][k]!=-inf:                   # condition d'arrêt
        return DIST[i][j][k]
    else:
        L=[Bellman_rec2(M, i, j, k-1, DIST)]    # appel récursif
        for x in range(n):
            L.append(Bellman_rec2(M, i, x, k-1, DIST)+M[x][j])
                    # appel récursif
        DIST[i][j][k]=min(L)
        return DIST[i][j][k]

DIST=[[[-inf for k in range(n)] for j in range(n)] for i in range(n)]
          # toutes les distances valent -inf
départ=0                                        # sommet de départ
arrivée=1                                       # sommet d'arrivée
distance3=Bellman_rec2(M, départ, arrivée, n-1, DIST)
print("Distance parcourue Bellman_rec2 :", distance3)
```

**d)** L'algorithme `Bellman_iterative` utilise la programmation dynamique avec la technique « *Bottom Up* » : on utilise la même formule de récurrence mais on part de la plus petite valeur de $k$ ($k = 0$) au lieu de partir de la plus grande valeur de $k$ ($k = n-1$, technique « *Top Down* »).

Il faut trois boucles `for` imbriquées pour faire varier $i$, $j$ et $k$.

- Dans la technique « *Top Down* », on ne traite que les sous-problèmes nécessaires. On n'a pas besoin de remplir entièrement la liste `DIST` pour obtenir la solution optimale au problème.
- Dans la technique « *Bottom Up* », on traite tous les sous-problèmes (trois boucles `for` imbriquées). Il faut d'abord remplir entièrement la liste `DIST` avant de retourner la solution optimale au problème.

```
def Bellman_iterative (M, départ, arrivée, DIST):
    # la fonction retourne DIST[départ][arrivée][n-1]
    # n = nombre de sommets
    n=len(M)                    # nombre de lignes de M
    for k in range (n):         # k varie entre 0 inclus et n exclu
        for i in range(n):      # i varie entre 0 inclus et n exclu
            for j in range(n):  # j varie entre 0 inclus et n exclu
                if k==0:
                    if j==i:
                        DIST[i][i][k]=0
                    else:
                        DIST[i][j][k]=inf
                else:
                    L=[DIST[i][j][k-1]]
                    for x in range(n):
```

```
                        L.append(DIST[i][x][k-1]+M[x][j])
                    DIST[i][j][k]=min(L)
    return DIST[départ][arrivée][n-1]

DIST=[[[-inf for k in range(n)] for j in range(n)] for i in range(n)]
          # toutes les distances valent -inf
départ=0                        # sommet de départ
arrivée=1                       # sommet d'arrivée
distance4=Bellman_iterative(M, départ, arrivée, DIST)
print("Distance parcourue Bellman_iterative :", distance4)
```

**e)** On définit une liste `PRECEDENT` : $\text{PRECEDENT}[i][j][k]$ est le sommet précédent $j$ sur un chemin optimal de $i$ à $j$ qui emprunte au plus $k$ arcs.

- $i$ varie entre 0 et $n-1$ inclus.
- $j$ varie entre 0 et $n-1$ inclus.
- $k$ varie entre 0 et $n-1$ inclus.

Toutes les valeurs de `PRECEDENT` sont initialisées à $-1$.

- Si $i \neq j$ et si $-\infty < \text{DIST}[i][j][k] < \infty$, alors $\text{PRECEDENT}[i][j][k]$ est le sommet précédent $j$ sur un chemin optimal de $i$ à $j$ qui emprunte des sommets intermédiaires d'indice strictement inférieur à $k$.
- $\text{PRECEDENT}[i][j][1] = i$ si $i \neq j$ et $\text{M}[i][j] < \infty$. On connaît le sommet précédent $j$ puisque le chemin emprunte au plus 1 arc.
- La valeur de $\text{PRECEDENT}[i][j][k]$ lorsque $i = j$ ou $\text{DIST}[i][j][k] = \infty$ ou $\text{DIST}[i][j][k] = -\infty$ n'a aucune importance. On peut la prendre égale à $-1$.
- $\text{PRECEDENT}[i][j][k] = \text{PRECEDENT}[i][j][k-1]$ si $d_k(i,j) = d_{k-1}(i,j)$ et $-\infty < d_k(i,j) < \infty$.

On obtient le chemin suivi en utilisant `PRECEDENT[départ][i][n-1]` puisqu'on a parcouru au plus $n-1$ arcs pour aller de `départ` à `arrivée`.

```
def Bellman_iterative2 (M, départ, arrivée, DIST, PRECEDENT):
    # la fonction retourne DIST[départ][arrivée][n-1]
    # n = nombre de sommets
    # PRECEDENT[i][j][k] = sommet précédent j sur un chemin optimal
    # de i à j qui emprunte au plus k arcs
    n=len(M)                    # nombre de lignes de M
    for k in range (n):         # k varie entre 0 inclus et n exclu
        for i in range(n):      # i varie entre 0 inclus et n exclu
            for j in range(n):  # j varie entre 0 inclus et n exclu
                if k==0:
                    if j==i:
                        DIST[i][i][k]=0
                    else:
                        DIST[i][j][k]=inf
                else:
                    L=[DIST[i][j][k-1]]
                    L2=[-1]     # on ne connaît pas le sommet précédent
                    for x in range(n):
                        L.append(DIST[i][x][k-1]+M[x][j])
```

```
                        L2.append(x)        # le sommet précédent j est x
                    # recherche du minimum et de l'indice
                    mini=L[0]
                    indice=0
                    for j1 in range(1,len(L)):
                        if L[j1]<mini:
                            mini=L[j1]      # valeur du minimum
                            indice=L2[j1]   # sommet précédent j
                    if indice >0:
                        PRECEDENT[i][j][k]=indice
                    DIST[i][j][k]=mini
                    if DIST[i][j][k]==DIST[i][j][k-1] \
                        and DIST[i][j][k-1]>-inf and DIST[i][j][k-1]<inf:
                        PRECEDENT[i][j][k]=PRECEDENT[i][j][k-1]
    return DIST[départ][arrivée][n-1]

DIST=[[[-inf for k in range(n)] for j in range(n)] for i in range(n)]
          # toutes les distances valent -inf
PRECEDENT=[[[-inf for k in range(n)] for j in range(n)] for i in range(n)]
          # tous les éléments valent -1
for i in range(n):
    for j in range(n):
        if i!=j and M[i][j]<inf:
            PRECEDENT[i][j][1]=i

départ=0                # sommet de départ
arrivée=1               # sommet d'arrivée
distance5=Bellman_iterative2(M, départ, arrivée, DIST, PRECEDENT)
print("Distance parcourue Bellman_iterative2 :", distance5)

# recherche du chemin suivi entre les sommets départ et arrivée
indice_boucle=0
i=arrivée
chemin=[arrivée]        # initialisation de la liste des sommets parcourus
while (i!=départ) and indice_boucle<n:
    i=int(PRECEDENT[départ][i][n-1])
    chemin.append(i)
    indice_boucle=indice_boucle+1
# on obtient la liste des sommets en sens inverse
chemin.reverse()        # il faut inverser la liste chemin
print("Chemin suivi :",chemin)
```

**f)** On modifie la fonction récursive `Bellman_rec2`. La fonction récursive `Bellman_rec3` retourne `DIST[i][j][k]` et `PRECEDENT[i][j][k]`. On cherche $\min_{0\leq x<n}\left(d_{k-1}(i,j), d_{k-1}(i,x)+M[x][j]\right)$.

- Lorsqu'on appelle `Bellman_rec3(M, i, j, k-1, DIST, PRECEDENT)`, on récupère `DIST[i, j, k-1]` et `PRECEDENT[i][j][k-1]`. On ajoute `DIST[i][j][k-1]` dans `L` et `PRECEDENT[i][j][k-1]` dans `L2`.

- Lorsqu'on appelle `Bellman_rec3(M, i, x, k-1, DIST, PRECEDENT)`, on récupère `DIST[i][x][k-1]` que l'on ajoute dans `L`. Le sommet précédent *j* est *x*. On ajoute *x* dans `L2`.

On parcourt entièrement la liste `L` pour récupérer le minimum de ses éléments (`mini`) ainsi que l'indice (`ind_mini`) correspondant au minimum. `L2[ind_mini]` représente le sommet précédent *j*.

On a alors : `DIST[i][j][k]=mini` et `PRECEDENT[i][j][k]=L2[ind_mini]`.

**Recherche du chemin suivi** :

On ne peut pas utiliser `PRECEDENT[départ][i][n-1]` dans la question f) car dans la technique « *Top Down* » on n'a pas rempli entièrement les listes `DIST` et `PRECEDENT`. Par contre, dans la technique « *Bottom Up* » de la question e), les listes `DIST` et `PRECEDENT` sont remplies entièrement.

- On appelle la fonction récursive `Bellman_rec3(M, départ, arrivée, n-1, DIST, PRECEDENT)`. On récupère `PRECEDENT[i][j][k]` avec `i=départ`, `j=arrivée` et `k=n-1`. On récupère le sommet précédent, ici le sommet 3.
- Pour obtenir le sommet précédent au sommet 3, la fonction `Bellman_rec3` a appelé récursivement `Bellman_rec3` avec $i = 0$, $j = 3$ et $k = (n–1)–1$.
- `i=int(PRECEDENT[départ][i][n-1-indice_boucle])` permet d'obtenir le sommet précédent *i*. On utilise un arc en moins à chaque fois que l'on remonte d'un sommet supplémentaire en partant du sommet `arrivée`.

```
def Bellman_rec3(M, i, j, k, DIST, PRECEDENT):
    # la fonction retourne DIST[i][j][k] et PRECEDENT[i][j][k]
    # pour la matrice M ; n = nombre de sommets
    n=len(M)                              # nombre de lignes de M
    if k==0:                              # condition d'arrêt
        if j==i:
            return 0, -1
        else:
            return inf, -1
    elif DIST[i][j][k]!=-inf:             # condition d'arrêt
        return DIST[i][j][k], PRECEDENT[i][j][k]
    else:
        a, b=Bellman_rec3(M, i, j, k-1, DIST, PRECEDENT)     # appel récursif
        L, L2=[a], [b]
        for x in range(n):
            L.append(Bellman_rec3(M, i, x, k-1, DIST, PRECEDENT)[0]+M[x][j])
                    # appel récursif
            L2.append(x) #le sommet précédent j est x
        # recherche du minimum et de l'indice
        mini=L[0]
        ind_mini=0                        # indice du minimum de L
        for j1 in range(1, len(L)):
            if L[j1]<mini:
                mini=L[j1]                # valeur du minimum
                ind_mini=j1               # indice correspondant au minimum de L
        DIST[i][j][k]=mini
        PRECEDENT[i][j][k]=L2[ind_mini]  # sommet précédent j
        return DIST[i][j][k], PRECEDENT[i][j][k]
```

```
DIST=[[[-inf for k in range(n)] for j in range(n)] for i in range(n)]
        # toutes les distances valent -inf
PRECEDENT=[[[-inf for k in range(n)] for j in range(n)] for i in range(n)]
        # tous les éléments valent -1
for i in range(n):
    for j in range(n):
        if i!=j and M[i][j]<inf:
            PRECEDENT[i][j][1]=i


départ=0                    # sommet de départ
arrivée=1                   # sommet d'arrivée
distance6=Bellman_rec3(M, départ, arrivée, n-1, DIST, PRECEDENT) [0]
print("Distance parcourue Bellman_rec3 :", distance6)
# recherche du chemin suivi entre les sommets départ et arrivée
indice_boucle=0
i=arrivée
chemin=[arrivée]            # initialisation de la liste des sommets parcourus
while (i!=départ) and indice_boucle<n:
    i=int(PRECEDENT[départ][i][n-1-indice_boucle])
    chemin.append(i)
    indice_boucle=indice_boucle+1
# on obtient la liste des sommets en sens inverse
chemin.reverse()            # il faut inverser la liste chemin
print('Chemin suivi :', chemin)
```

**g)** La programmation dynamique est souvent utilisée pour résoudre des problèmes d'optimisation. Elle comprend plusieurs étapes :

- Recherche d'une récurrence pour déterminer la valeur d'une solution optimale.
- Utilisation de la technique *Top Down* ou *Bottom Up*.

La programmation dynamique et la méthode gloutonne reposent sur l'utilisation de sous-problèmes. Il y a une grosse différence :

- Dans la programmation dynamique, on calcule toutes les solutions des sous-problèmes que l'on combine pour obtenir une solution optimale.
- Dans la méthode gloutonne, on choisit une solution qui semble être la meilleure et on résout le sous-problème qui en résulte.

On peut envisager plusieurs algorithmes pour déterminer un plus court chemin entre un sommet `départ` et un sommet d'arrivée `arrivée` :

- **Approche naïve** : on explore tous les chemins possibles et on choisit celui qui a la distance la plus petite. Cet algorithme repasse très souvent par d'anciens sommet et on parcourt le graphe de très nombreuses fois.
- **Algorithme de Dijkstra** : on définit un sous-graphe $G'$. À chaque étape, on ajoute un nouveau sommet `position` dans $G'$ en cherchant la distance minimale entre `début` et `position`. On considère $d(\text{depart}, \text{position}) + M[\text{position}][i]$ pour déterminer le nouveau sommet `position`.

  On réalise un choix local optimal qui semble être le meilleur. On ne reviendra plus sur ce choix dans les étapes ultérieures.

Il s'agit d'une méthode gloutonne. On peut montrer que cette méthode donne la solution optimale mais elle ne s'applique pas si le graphe possède des arcs de poids négatif.

- **Algorithme de Bellman-Ford** : à chaque étape, on calcule $d_k(i,j) = \min_{0 \le x < n} \left( d_{k-1}(i,j), d_{k-1}(i,x) + M[x][j] \right)$

  (ressemblance avec l'algorithme de Dijkstra où on calcule $d(\text{depart}, \text{position}) + M[\text{position}][i]$). **On recalcule toutes les distances à l'étape suivante** puisqu'on a un arc supplémentaire et l'arête de cet arc peut être de poids négatif (on dit qu'un arc n'est pas relâché définitivement car on peut avoir des arêtes de poids négatif). Il s'agit donc d'un algorithme qui utilise la **programmation dynamique**.

  Dans la méthode gloutonne, on ne revient pas sur un choix local optimal alors qu'avec la programmation dynamique on peut revenir sur les choix précédents.

**Remarque**

On suppose que l'on n'a pas de cycle de poids négatif. Sinon on peut trouver une boucle infinie et à chaque étape on peut trouver un meilleur chemin (on améliore la distance parcourue) en empruntant ce cycle qui diminue la distance parcourue.

**h)** On ajoute une étape supplémentaire dans la boucle `for k in range(n):` du programme de la question e). On peut montrer que l'algorithme de Bellman-Ford s'arrête après au plus $k = n-1$ étapes de cette boucle qui modifie les éléments $d_k(i,j)$ ($i$ varie entre 0 et $n-1$ ; $j$ varie entre 0 et $n-1$).

Si on exécute une fois de plus cette boucle et que les distances entre $i$ et $j$ sont modifiées, alors le graphe possède un cycle de poids négatif.

Pour détecter si les distances $i$ et $j$ sont modifiées entre l'étape $n-1$ et l'étape $n$, on exécute le test `DIST[i][j][n]!=DIST[i][j][n-1]` pour toutes les valeurs possibles de $i$ et $j$.

```
def Bellman_iterative3 (M, départ, arrivée, DIST, PRECEDENT):
    # la fonction retourne DIST[départ][arrivée][n-1]
    # n = nombre de sommets
    # PRECEDENT[i][j][k] = sommet précédent j sur un chemin optimal
    # de i à j qui emprunte au plus k arcs
    n=len(M)                            # nombre de lignes de M
    for k in range(n+1):                # k varie entre 0 inclus et n+1 exclu
        #on fait une étape supplémentaire avec k=n
        for i in range(n):              # i varie entre 0 inclus et n exclu
            for j in range(n):          # j varie entre 0 inclus et n exclu
                if k==0:
                    if j==i:
                        DIST[i][i][k]=0
                    else:
                        DIST[i][j][k]=inf
                else:
                    L=[DIST[i][j][k-1]]
                    L2=[-1]             # on ne connaît pas le sommet précédent
                    for x in range(n):
                        L.append(DIST[i][x][k-1]+M[x][j])
                        L2.append(x)    # le sommet précédent j est x
```

```
                    # recherche du minimum et de l'indice
                    mini=L[0]
                    indice=0
                    for j1 in range(1,len(L)):
                        if L[j1]<mini:
                            mini=L[j1]      # valeur du minimum
                            indice=L2[j1]   # sommet précédent j
                    if indice >0:
                        PRECEDENT[i][j][k]=indice
                    DIST[i][j][k]=mini
                    if DIST[i][j][k]==DIST[i][j][k-1]\
                      and DIST[i][j][k-1]>-inf and DIST[i][j][k-1]<inf:
                        PRECEDENT[i][j][k]=PRECEDENT[i][j][k-1]
    return DIST[départ][arrivée][n-1]

DIST=[[[-inf for k in range(n+1)] for j in range(n)] for i in range(n)]
          # toutes les distances valent -inf
PRECEDENT=[[[-inf for k in range(n+1)] for j in range(n)]\
       for i in range(n)]
            # tous les éléments valent -1
départ=0                                # sommet de départ quelconque
arrivée=0                               # sommet d'arrivée quelconque
distance7=Bellman_iterative3(M, départ, arrivée, DIST, PRECEDENT)
# recherche d'une case différente k=n-1 et k=n
rep=False
for i in range(n):
    for j in range(n):
        if DIST[i][j][n]!=DIST[i][j][n-1]:
            rep=True
if rep==False:
    print("Le graphe ne possède pas de cycle de poids négatif.")
else:
    print("Le graphe possède au moins un cycle de poids négatif.")
```

# Intelligence artificielle (spé)

CHAPITRE 14

Plan



### Thèmes abordés dans les exercices

- Algorithme des *k* plus proches voisins avec distance euclidienne
- Algorithme des *k*-moyennes
- Algorithme min-max avec une heuristique

### Points essentiels du cours pour la résolution des exercices

- Matrice de confusion
- Apprentissage supervisé et non supervisé
- Valeur optimale du nombre total de clusters

## Les méthodes à retenir

Le *Machine Learning* est une branche de l'intelligence artificielle qui permet à un ordinateur d'apprendre à partir de données. On distingue deux types de problèmes :

- **L'apprentissage supervisé**. On cherche une fonction de prédiction à partir d'une base d'apprentissage. Exemple : régression linéaire, algorithme des *k* plus proches voisins.
- **L'apprentissage non supervisé**. Les données ne sont pas étiquetées. Il faut découvrir des groupes. Le programme regroupe les données par groupes, ou clusters (application au regroupement de clients ayant un comportement similaire afin d'adapter les annonces pour le marketing). Exemple : algorithme des *k*-moyennes.

### Apprentissage supervisé – Algorithme des *k* plus proches voisins

L'algorithme des *k* plus proches voisins est basé sur un algorithme d'apprentissage à partir d'observations étiquetées. Le modèle prédictif est utilisé dans plusieurs cas :

- **Régression** : le résultat est un réel. Le résultat est la moyenne des valeurs des *k* plus proches voisins. Exemple : prédiction de la solubilité d'une molécule dans l'eau en mg/mL.

- **Classification** : le résultat obtenu est une classe d'appartenance (0, 1, 2, …, $C$−1) si on considère $C$ classes possibles. Dans l'exemple qui suit, on a $C = 3$ classes (0 pour *setosa*, 1 pour *versicolor* et 2 pour *virginica*).
- **Classification binaire** : le résultat obtenu est 0 ou 1. Exemples : l'email reçu est-il un spam ? La photo est-elle celle d'un chat ? On utilise un nombre impair de voisins pour ne pas avoir d'*ex aequo*.

On considère une matrice de données `data` contenant $n$ lignes et 3 colonnes. Chaque ligne contient les caractéristiques d'un iris : longueur des pétales (en cm), largeur des pétales (en cm) et désignation de l'espèce de l'iris (0 pour *setosa*, 1 pour *versicolor* et 2 pour *virginica*). On utilise les listes de listes pour représenter les matrices dans Python. La méthode `A.sort()` permet de trier en place une liste de listes `A` en fonction du premier élément de chaque liste interne.

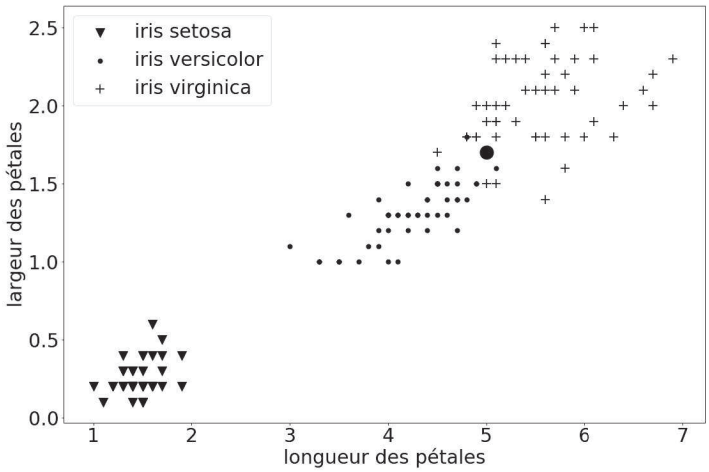
La fonction `distance` admet comme arguments deux listes `ptA` et `ptB`. Elle retourne la distance euclidienne entre le point $A(x_A, y_A)$ et le point $B(x_B, y_B)$ : $\text{distance} = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$.

La fonction `knn` admet comme arguments une matrice `data` ($n$ lignes et 3 colonnes), une liste `iris2` et un entier $k$. La fonction retourne une prédiction de l'espèce de `iris2` (liste de deux valeurs : longueur et largeur des pétales). Les étapes de l'algorithme des $k$ plus proches voisins sont les suivantes :

- Pour chaque iris (noté `iris1`) de `data`, on calcule la distance euclidienne entre `iris1` et `iris2`.
- On définit une matrice `dist` ($n$ lignes et deux colonnes). Chaque ligne contient la distance euclidienne entre `iris1` et `iris2` ainsi que la désignation de l'espèce de `iris1`.
- On trie la liste de listes `dist` dans l'ordre croissant des distances euclidiennes entre `iris1` et `iris2`.
- On en déduit une prédiction de l'espèce de `iris2` en cherchant la désignation majoritaire parmi les $k$ plus proches voisins de `iris2`.

```
def distance(ptA, ptB):            # distance euclidienne entre ptA et ptB
    # ptA est une liste de deux éléments, ptB est une liste de deux éléments
    return ((ptB[0]-ptA[0])**2+(ptB[1]-ptA[1])**2)**(0.5)


def knn(data, iris2, k):
    # calcul de la distance de iris2 à tous les points de data
    # la fonction retourne une prédiction de l'espèce de l'iris
    # algorithme des k(int) plus proches voisins
    # iris2 : liste de deux valeurs : longueur et largeur des pétales
    n=len(data)                    # nombre de lignes de data
    dist=[[0 for j in range(2)] for i in range(n)]
            # matrice avec distance et numéro espèce
    for i in range(len(data)):     # i varie entre 0 inclus et len(data) exclu
        iris1=[data[i][0],data[i][1]]
        dist[i][0]=distance(iris1, iris2)
        dist[i][1]=data[i][2]
            # on récupère la distance et la désignation de l'espèce
    dist.sort()                    # tri par ordre croissant des distances
    # on récupère la liste des distances triées par ordre croissant
    # choix de l'espère
    list_nb_espece=[[0, i] for i in range(3)]
            # liste de listes : (total, numéro de l'espèce)
    for i in range(k):             # i varie entre 0 inclus et k exclu
        indice=int(dist[i][1])     # dist[i][1] : numéro de l'espèce
        list_nb_espece[indice][0]+=1
    list_nb_espece.sort() #tri de list_nb_espece par ordre croissant de total
```

```
    predict=list_nb_espece[2][1]    # le dernier élément est celui
                                    # qui apparaît le plus
    return predict
```




On applique l'algorithme des *k* plus proches voisins à `pt_search=[5, 1.7]` avec $k = 5$.

À la fin de l'exécution de la fonction `knn`, la liste `list_nb_espece` vaut : [[0, 0], [2, 1], [3, 2]].

Parmi les 5 plus proches voisins, on 3 voisins *virginica*, 2 voisins *versicolor* et aucun *setosa*.

Le programme Python renvoie : 2 *virginica*. L'iris recherché (gros cercle noir sur le graphique ci-dessus) est plus près des iris *virginica* que des iris *versicolor* et *setosa*.

# Apprentissage supervisé – Algorithme des *k*-moyennes

L'algorithme des *k*-moyennes permet de trouver des groupes (appelés clusters) parmi un nuage de points. On utilise deux listes `X` et `Y` : chaque point *i* est caractérisé par son abscisse `X[i]` et son ordonnée `Y[i]`. Le but est de regrouper les éléments qui se « ressemblent » dans *K* clusters.

La fonction `initcluster` admet comme arguments un entier `K` et deux listes `X`, `Y`. Cette fonction retourne une liste de listes contenant les coordonnées de *K* points différents tirés aléatoirement parmi le nuage de points.

**Initialisation** :

On choisit au hasard *K* centres des clusters dans un nuage de points. Chaque centre est caractérisé par une abscisse `X[i]` et une ordonnée `Y[i]`.

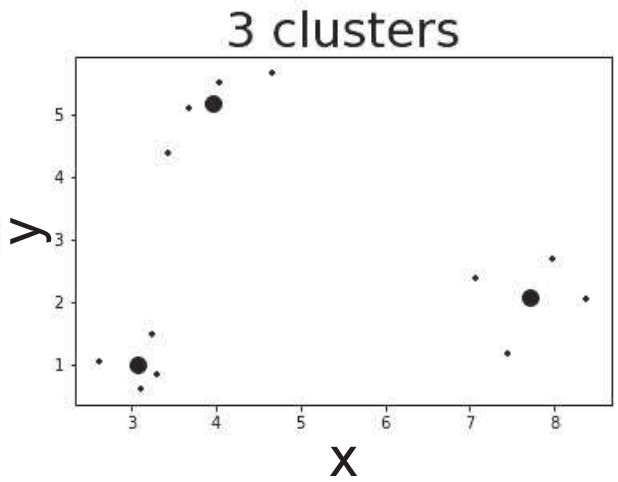
**Boucle tant que les points changent de cluster** :

- Placer chaque point dans le cluster *k* qui lui est le plus proche.
- Recalculer les centres des clusters (appelés également centroïdes). On utilisera la moyenne des abscisses et des ordonnées des points appartenant à un cluster.

La fonction `algokm` admet comme arguments un entier `K` et deux listes `X`, `Y`. Cette fonction retourne une liste de listes contenant les coordonnées des centres des *K* clusters et la liste `A` (`A[i]` désigne le numéro du cluster du point `[X[i],Y[i]]`).

```
import random as rd               # module random renommé rd
inf=1e10                          # variable représentant l'infini

def initcluster(K, X, Y):
    # la fonction retourne une liste de listes contenant les coordonnées
    # des centres des K clusters tirés aléatoirement parmi le nuage de points
    # arguments d'entrée : entier K et deux listes X et Y
    n=len(X)
    L=[]                           # liste de listes : coordonnées des clusters
    liste_indice=[]
    while len(L)!=K:
        i=rd.randint(0, n-1)
        # indice aléatoire compris entre 0 inclus et n-1 inclus
        if i not in liste_indice:
            liste_indice.append(i)
            L.append([X[i],Y[i]])  # ajoute l'abscisse et l'ordonnée d'un point
    return L                       # liste des K clusters

def algokm (K, X, Y):
    # la fonction retourne une liste de listes contenant les coordonnées
    # des centres des K clusters avec l'algorithme des k-moyennes
    # A : liste d'affectation des points i à un cluster
    # arguments d'entrée : entier K = nombre de clusters,
    # X et Y deux listes représentant les abscisses et ordonnées des points
    # X[i] = abscisse du point d'indice i et Y[i] = ordonnée du point d'indice i

    # initialisation des K centres des clusters
    L=initcluster(K, X, Y)
    A=[0 for i in range(N)]  # liste pour affecter les points à un cluster
                             # A[i] : numéro du cluster pour le point X[i],Y[i]
    flag_stable=False #flag à False si on affecte un point à un autre cluster

    while flag_stable==False:
        flag_stable=True     # flag initialisé à True
                             # il passe à False si le point i change de cluster
        # on place chaque point dans le cluster k le plus proche
        for i in range(N):   # i varie entre 0 inclus et N exclu
            val_min=inf      # initialisation de val_min à infini
            ind_min=0        # indice du cluster correspondant au minimum
            for k in range(K):      # k varie entre 0 inclus et K exclu
                if distance([X[i], Y[i]], L[k])<val_min:
                    val_min=distance([X[i],Y[i]],L[k])
                    ind_min=k
            if A[i]!=ind_min:
                A[i]=ind_min
                flag_stable=False   # le point i a changé de cluster
```

```
        if flag_stable==False or K==1:   # teste si on affecte un point
                                         # à un autre cluster
            # on recalcule les centres de chaque cluster
            som_x=[0 for i in range(K)]
            som_y=[0 for i in range(K)]
            nb_el=[0 for i in range(K)]
            for i in range(N):
                k=int(A[i])          # le point i appartient au cluster k
                som_x[k]=som_x[k]+X[i]
                som_y[k]=som_y[k]+Y[i]
                nb_el[k]+=1          # on ajoute un point de plus à ce cluster
            for k in range(K):
                if nb_el[k]==0:      # teste si aucun point dans le cluster k
                    abscisse=0
                    ordonnée=0
                else:
                    abscisse=som_x[k]/nb_el[k]
                    ordonnée=som_y[k]/nb_el[k]
                L[k]=[abscisse, ordonnée]
    return L, A
```




# Intelligence artificielle et jeux

L'intelligence artificielle est « l'ensemble des théories et des techniques mises en œuvre en vue de réaliser des machines capables de simuler l'intelligence ».

### Principe de l'algorithme min-max pour le jeu de morpion

Le morpion est un jeu à somme nulle (les gains d'un joueur sont l'opposé des gains de l'autre joueur) avec un nombre fini de stratégies. Le meilleur gain possible pour le joueur 1 est +1 (victoire pour le joueur 1 et défaite pour le joueur 2) et le meilleur gain pour le joueur 2 est −1 (défaite pour le joueur 1 et victoire pour le joueur 2).

- Le joueur 1, que l'on appellera MAX, pose les pions noirs.
- Le joueur 2, que l'on appellera MIN, pose les pions blancs.

Le joueur 2 cherche à minimiser ses gains alors que le joueur 1 cherche à maximiser ses gains.

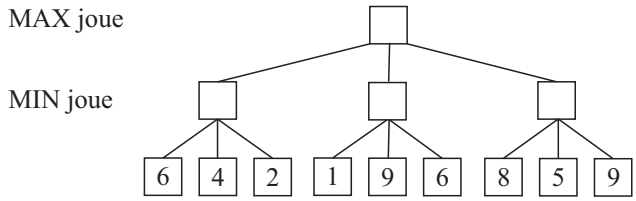
L'algorithme min-max est utilisé dans de nombreux jeux : Othello, échecs…

### Principe de l'algorithme min-max dans le cas général

On considère le cas général d'un jeu à deux joueurs à somme nulle et avec un nombre fini de stratégies.

À un moment donné du jeu, c'est au joueur 1 (MAX) de jouer. On suppose qu'il a trois possibilités. On représente sur l'arbre de jeu ci-dessous les trois possibilités. Ensuite c'est au joueur 2 (MIN) de jouer. On suppose qu'il a également trois possibilités. On représente sur le graphe ci-dessous la valeur du gain quand MAX et MIN ont joué.

Chaque nœud correspond à une position de jeu.



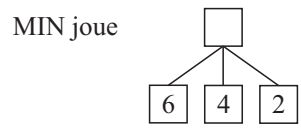

Le nœud du niveau supérieur est appelé racine.

Les feuilles sont les nœuds terminaux pour lesquels ne partent aucune branche et correspondent souvent à une fin de partie. La hauteur de l'arbre ci-dessus est 3. Pour les jeux, on parle plutôt de profondeur de jeu que de hauteur. La profondeur pouvant être très importante pour arriver à une fin de partie, on limite souvent la profondeur d'étude (voir exercice 14.3 « Jeu de morpion, algorithme min-max »). Dans ce cas, on ne sait pas si les feuilles correspondent à une victoire ou à une défaite. On définit alors une fonction d'évaluation appelée GAIN qui évalue le gain pour chaque nœud. Comme on considère un jeu à somme nulle, le joueur 1 (MAX) cherche à maximiser ses gains alors que le joueur 2 (MIN) cherche à les minimiser à chaque coup.

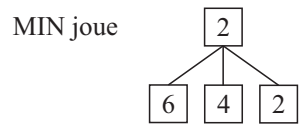
Le joueur MAX a trois possibilités pour jouer et choisit le coup qui va maximiser ses gains. Lorsque le jouer MAX a joué, le joueur MIN va jouer en cherchant à minimiser ses gains.

On parcourt en profondeur cet arbre en utilisant la fonction GAIN pour déterminer le meilleur coup à jouer pour MAX.

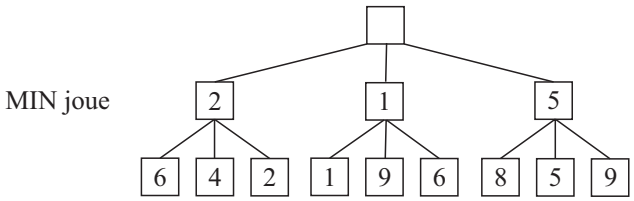
On considère les feuilles : 6, 4 et 2. On remonte dans l'arbre. C'est à MIN de jouer.



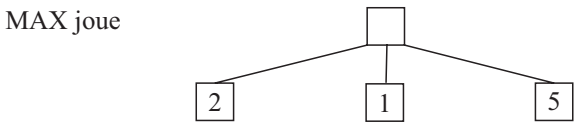

Il choisit le coup qui minimise le gain. Il choisit alors le coup avec un gain égal à 2.



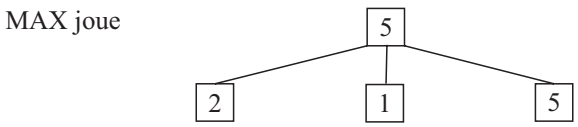
On complète l'arbre de jeu avec le minimum des gains.



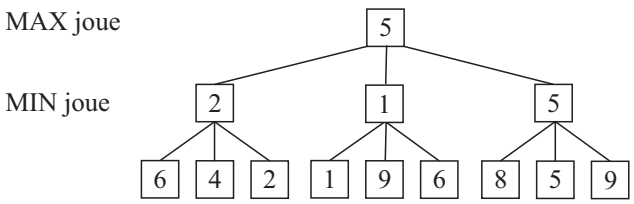

On remonte dans l'arbre et on se pose la question : Quel coup choisir pour le joueur 1 (MAX) ?




Le joueur 1 (MAX) cherche à maximiser ses gains. Il va donc choisir le coup avec un gain égal à 5.




On obtient alors l'arbre de jeu :




Plus la profondeur est grande, plus le coup choisi sera de meilleure qualité.

# Énoncés des exercices

## 14.1

**Algorithme des *k* plus proches voisins**

On considère un jeu de données « fic1.csv » contenant *n* lignes. Chaque ligne contient les caractéristiques des reptiles : longueur (en m), taille de la gueule (en m) et désignation du reptile (0 pour alligator, 1 pour crocodile). Les données sont séparées avec le séparateur « `;` ». On utilise les listes de listes pour représenter les matrices dans Python. La méthode `A.sort()` permet de trier en place une liste de listes `A` en fonction du premier élément de chaque liste interne.

Rappels 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()        # récupération de l'ensemble des données
                     # du fichier dans une liste
\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 la chaîne de caractères 'exemple'
f.close()            # ferme le fichier
```

**a)** Écrire une fonction `lect_data` qui admet comme argument un nom de fichier et retourne une matrice contenant *n* lignes et trois colonnes : longueur du reptile, taille de la gueule et désignation du reptile (0 ou 1).

**b)** Écrire une fonction `calc_dist` qui admet comme arguments deux listes `ptA` et `ptB`. La fonction retourne la distance euclidienne entre le point *A* de coordonnées $\left(\text{ptA}[0], \text{ptA}[1]\right)$ et le point *B* de coordonnées $\left(\text{ptB}[0], \text{ptB}[1]\right)$.

**c)** Écrire une fonction `algoknn` qui admet comme arguments une matrice `data` (*n* lignes et 3 colonnes), une liste `pt_search`, et un entier `k`. La fonction retourne une prédiction de l'espèce du reptile (noté `rept2`) caractérisé par `pt_search` (liste de deux valeurs : longueur et taille de la gueule).

Les étapes de l'algorithme des *k* plus proches voisins sont les suivantes :

- Pour chaque reptile (noté `rept1`) de `data`, on calcule la distance euclidienne entre `rept1` et `rept2`.
- On définit une liste `mat_dist` (*n* lignes et deux colonnes). Chaque ligne contient la distance euclidienne entre `rept1` et `rept2` ainsi que la désignation de l'espèce de `rept1`.
- On trie la liste de listes `mat_dist` dans l'ordre croissant des distances euclidiennes entre `rept1` et `rept2`.
- On en déduit une prédiction de l'espèce de `rept2` en cherchant la désignation majoritaire parmi les *k* plus proches voisins de `rept2`.

**d)** Écrire le programme principal permettant d'afficher le nuage de points des données du fichier « fic1.csv » ainsi que le point recherché `pt_search` et d'afficher une prédiction de l'espèce du reptile2 caractérisé par `pt_search=[3.8, 0.32]` avec $k = 5$.

Le graphique doit avoir les caractéristiques suivantes :

- affichage de « longueur du reptile » pour l'axe des abscisses et « taille de la gueule » pour l'axe des ordonnées ;
- affichage en bleu avec marker « v » pour les alligators ;
- affichage en rouge avec marker « . » pour les crocodiles ;
- affichage de la légende « alligator » et « crocodile » ;
- affichage en noir avec `linewidth=8` pour le reptile `pt_search`.

On pourra utiliser `plt.scatter()` pour représenter un nuage de points en utilisant le module `matplotlib.pyplot` que l'on renomme `plt`. Les arguments d'entrée sont les mêmes que pour la fonction `plt.plot()`.

**e)** On considère les jeux de données « fic1.csv » (fichier d'apprentissage pour l'algorithme des $k$ plus proches voisins) et « fic2.csv » (fichier de test). Écrire une fonction `evalKNN` qui admet comme arguments `data1` (liste d'apprentissage), `data2` (liste de test) et un entier `k`. La fonction retourne la matrice suivante contenant 2 lignes et 2 colonnes :

| | | **Classe réelle** | |
|---|---|---|---|
| | | alligator | crocodile |
| **Classe prédite** | alligator | … | … |
| | crocodile | … | … |

`mat[0][1]` représente le nombre de fois où l'algorithme prédit « alligator » alors que l'espèce est en réalité « crocodile ».

**f)** On définit la matrice de confusion `matconf` pour le crocodile avec $k = 2$ :

| | | **Classe réelle** | |
|---|---|---|---|
| | | crocodile | non crocodile |
| **Classe prédite** | crocodile | True Positives (TP) | False Positives (FP) |
| | non crocodile | False Negatives (FN) | True Negatives (TN) |

`matconf [0][1]` représente le nombre de faux positifs (FP) dans le jeu de test.

Que représentent les indicateurs suivants ? :

- $rappel = \dfrac{TP}{TP + FN}$
- $spécificité = \dfrac{TN}{TN + FP}$
- $précision = \dfrac{TP}{TP + FP}$

Écrire une fonction `matconf_indicateurs` qui admet comme arguments `data1` (liste d'apprentissage) et `data2` (liste de test). La fonction retourne la matrice de confusion pour le crocodile avec $k = 2$, `rappel`, `spécificité` et `précision`.

**g)** Écrire une fonction `predict_k` qui admet comme arguments `data1` (liste d'apprentissage) et `data2` (liste de test). La fonction retourne la valeur optimale de $k$ en utilisant les étapes suivantes :

- Pour toutes les valeurs de *k* possibles, on applique l'algorithme des *k* plus proches voisins à toutes les espèces du fichier de test.
- On calcule pour chaque valeur de *k* le pourcentage d'espèces mal prédites du fichier de test.
- On en déduit la valeur optimale de *k* correspondant au pourcentage d'espèces mal prédites le plus faible.

## 14.2

■■□□

### Algorithme des *k*-moyennes

L'algorithme des *k*-moyennes permet de regrouper des éléments qui se « ressemblent » dans des groupes (appelés clusters). On considère les données client d'un magasin de gros. Pour chaque ligne du fichier « clients_data.csv » (voir site Dunod), on a les données suivantes : canal du client (1 : hôtel/restaurant, 2 : vente au détail), région (1 : Lisbonne, 2 : Porto, 3 : autre région), frais, lait, épicerie, surgelé, détergent, charcuterie. Les données sont séparées avec le séparateur « , ». Les ventes annuelles des 6 catégories sont en unités monétaires. Les trois premières lignes du fichier .csv sont les suivantes :

```
Channel,Region,Fresh,Milk,Grocery,Frozen,Detergents_Paper,Delicassen
2,3,12669,9656,7561,214,2674,1338
2,3,7057,9810,9568,1762,3293,1776
```

Rappels 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()        # récupération de l'ensemble des données
                     # du fichier dans une liste
\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 la chaîne de caractères 'exemple'
f.close()            # ferme le fichier
```

On utilise les listes de listes pour représenter les matrices dans Python.

Rappels pour la génération de nombres aléatoires :

```
import random as rd # module random renommé rd
rd.randint(a, b)    # renvoie un entier aléatoire M tel que a <= M <= b
```

**a)** Écrire une fonction `fic_data` qui admet comme argument un nom de fichier et retourne une matrice contenant *n* lignes et 6 colonnes correspondant aux catégories : frais, lait, épicerie, surgelé, détergent, charcuterie.

**b)** Écrire une fonction `distance` qui admet comme arguments deux listes `ptA` et `ptB`. La fonction retourne la distance euclidienne entre le point $A$ $\left(\text{ptA}[0], \text{ptA}[1], ..., \text{ptA}[5]\right)$ et le point $B$ $\left(\text{ptB}[0], \text{ptB}[1], ..., \text{ptB}[5]\right)$.

**c)** Écrire une fonction `initcluster` qui admet comme arguments un entier `K` et une matrice `data` (*n* lignes et 6 colonnes). Cette fonction retourne une liste de listes contenant les coordonnées de `K` points différents tirés aléatoirement parmi les *n* clients.

**d) Algorithme des *k*-moyennes**

**Initialisation** :

On choisit au hasard *K* centres des clusters parmi le fichier de clients. Chaque centre est caractérisé par un vecteur à 6 coordonnées (correspondant aux 6 catégories).

**Boucle tant que les clients changent de cluster** :

- Placer chaque client dans le cluster *k* qui lui est le plus proche.
- Recalculer les centres des clusters (appelés également centroïdes). On utilisera la moyenne de chaque catégorie des clients appartenant à un cluster.

On définit $W_K$ l'indicateur de compacité des clusters :

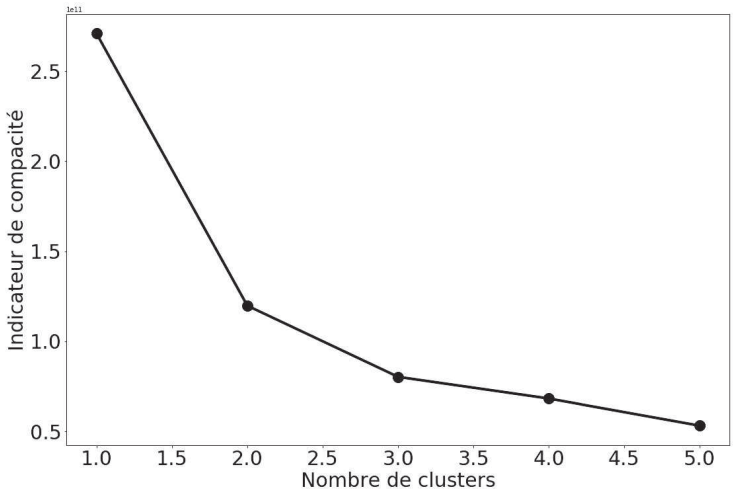
$W_K = \sum_{k=1}^{K} \sum_{i=1}^{n} \left(\text{distance}\left(\text{i-ème client}, \text{centre du k-ième cluster}\right)\right)^2$ en notant *K* le nombre total de clusters.

Écrire une fonction `algokm` qui admet comme arguments un entier `K` et une matrice `data` (*n* lignes et 6 colonnes). Cette fonction retourne une liste de listes contenant les coordonnées des centres des `K` clusters, l'indice de compacité et la liste `A` (`A[i]` désigne le numéro du cluster du client d'indice `i`). Quels sont les défauts de cet algorithme ?

**e)** Écrire le programme principal permettant :

- de représenter graphiquement $W_K$ en fonction de *K* (compris entre 1 et 5) ;
- d'afficher pour chaque cluster : le nombre de clients, la moyenne des ventes pour chaque catégorie, la moyenne des ventes.

**f)** On obtient le graphique suivant :




Expliquer comment la méthode du coude permet d'obtenir $K = 3$ la valeur optimale du nombre total de clusters.

On obtient alors les coordonnées des centres des 3 clusters : [[35941, 6044, 6288, 6713, 1039, 3049], [8296, 3787, 5162, 2582, 1724, 1138], [7751, 17910, 27037, 1970, 12104, 2185]].

La moyenne des ventes par cluster est : [59074, 22689, 68957]. Le nombre de clients par cluster est [60, 327, 53].

La stratégie marketing de l'entreprise est d'identifier des groupes de clients et de déployer une campagne de publicité en ciblant les catégories qui rapportent le moins pour les clients qui dépensent le plus. Détailler brièvement la campagne déployée par l'entreprise.

**g)** Quelle est la différence entre l'apprentissage supervisé et l'apprentissage non supervisé ?

**h)** Écrire le programme principal permettant de prédire le numéro d'un cluster pour un client dont on connaît les ventes pour les 6 catégories.

## 14.3

■■■□

### Jeu de morpion, algorithme min-max

On considère le jeu de morpion avec une grille 3×3. Les joueurs ajoutent au fur et à mesure un pion sur la grille en commençant par un pion noir. Les pions noirs sont représentés par « X » et les pions blancs par « O ». Les cases vides sont représentées par « . ». Le but est d'aligner 3 pions sur la grille.

On utilise la liste `jeu` pour représenter le plateau de jeu avec Python. Le plateau de jeu suivant est représenté par la liste `jeu=['.', '.', '.', 'O', '.', 'X', '.', 'X', '.']`. On repère une case par son indice. Par exemple l'indice 3 correspond à la case avec un pion blanc (« O »).

| | | |
|---|---|---|
| . | . | . |
| O | . | X |
| . | X | . |

**a)** Mise en place du jeu :

- Écrire une fonction `init` qui retourne une liste `jeu` correspondant à un plateau vide.
- Écrire une fonction `affiche` qui admet comme argument une liste `jeu` et qui permet d'afficher le plateau de jeu sur 3 lignes.
- Écrire une fonction `choixjoueur` qui admet comme argument une liste `jeu` et qui retourne le joueur (« `O` » ou « `X` ») devant jouer.
- Écrire une fonction `listecoups` qui admet comme argument une liste `jeu` et qui retourne une liste contenant les indices des cases vides.
- Écrire une fonction `gain` qui admet comme argument une liste `jeu` et qui retourne `1` si les noirs ont gagné, `-1` si les blancs ont gagné et `0` si aucun joueur n'a gagné même si la partie n'est pas terminée.

**b)** Écrire une fonction `jouercoup` qui admet comme arguments une liste `jeu` et un entier `coup`. Cette fonction retourne une nouvelle liste `jeu2` sans modifier la liste `jeu` en ajoutant un pion à l'indice `coup` de la liste `jeu`. On pourra utiliser la fonction `choixjoueur` pour déterminer quel joueur ajoute le pion à l'indice `coup`.

**c)** Principe de l'algorithme min-max :

- Écrire une fonction `valMax` qui admet comme argument la liste `jeu`. Cette fonction est appelée lorsque le joueur « X » veut choisir le meilleur coup afin de maximiser le gain sachant que le joueur « O » va le minimiser (on cherche le maximum des gains de `valMin(jeu)` sur tous les coups

possibles). Cette fonction retourne la valeur du maximum du gain et l'indice de la case à jouer (entier compris entre 0 et 9).

- Écrire une fonction `valMin` qui admet comme argument la liste `jeu`. Cette fonction est appelée lorsque le joueur « O » veut choisir le meilleur coup afin de minimiser le gain sachant que le joueur « X » va le maximiser (on cherche le minimum des gains de `valMin(jeu)` sur tous les coups possibles). Cette fonction retourne la valeur du minimum du gain et l'indice de la case jouée (entier compris entre 0 et 9).

Détail de l'algorithme `valMax(jeu)` :

- Si la partie est finie ou qu'un des joueurs a gagné, alors la fonction `valMax` retourne `gain(jeu), -1`.
- Sinon :
  - On parcourt tous les coups possibles. Pour chaque coup, on appelle la fonction `valMin` et on calcule le maximum de tous les gains.
  - La fonction retourne le gain maximum et l'indice du coup à jouer correspondant à ce gain.

Détail de l'algorithme `valMin(jeu)` :

- Si la partie est finie ou qu'un des joueurs a gagné, alors la fonction `valMin` retourne `gain(jeu), -1`.
- Sinon :
  - On parcourt tous les coups possibles. Pour chaque coup, on appelle la fonction `valMax` et on calcule le minimum de tous les gains.
  - La fonction retourne le gain minimum et l'indice du coup à jouer correspondant à ce gain.

**d)** Écrire une fonction `jouerminmax` qui admet comme argument une liste `jeu`. Cette fonction affiche les plateaux de jeu à chaque étape du jeu de morpion. L'ordinateur (pions noirs) joue contre l'ordinateur (pions blancs). Tant qu'un des joueurs n'a pas gagné et qu'il reste des coups à jouer, on utilise l'algorithme min-max pour choisir le coup suivant.

**e)** Écrire une fonction `jouercontreIA` qui admet comme argument une liste `jeu`. Cette fonction affiche les plateaux de jeu à chaque étape du jeu de morpion. L'ordinateur (ou IA = Intelligence Artificielle) a les pions noirs et l'humain a les pions blancs. Tant qu'un des joueurs n'a pas gagné et qu'il reste des coups à jouer, on utilise l'algorithme min-max pour choisir le coup suivant lorsque l'IA joue. L'humain tape au clavier l'indice de la case où il pose un pion blanc.

**f)** L'arbre de jeu peut devenir très grand et les appels récursifs peuvent être coûteux en mémoire. On définit un entier `profondeurmaxi` qui détermine la profondeur maximale explorée dans l'arbre de jeu. Écrire deux nouvelles fonctions récursives `valMax2` et `valMin2` qui admettent comme arguments une liste `jeu` et un entier `profondeur`. Le paramètre `profondeurmaxi` détermine le nombre de coups calculés à l'avance par l'algorithme.

**g)** Écrire une fonction `jouercontreIA3` qui admet comme arguments une liste `jeu` et un entier `profondeurmaxi`. Cette fonction permet à un humain de jouer contre l'ordinateur en choisissant la couleur des pions et un niveau de difficulté (l'entier `profondeurmaxi`). Par exemple : 2 pour un niveau débutant, 6 pour un niveau intermédiaire et 10 pour un niveau expert.

# Du mal à démarrer ?

## 14.1

**a)** Définir `n` le nombre de lignes du fichier. Écrire une boucle `for` en faisant varier `i` entre 0 inclus et `n` exclu. Récupérer une ligne du fichier et appliquer les fonctions `strip()` et `split(";")`.

**b)** Écrire une boucle `for` en faisant varier `k` entre 1 inclus et `n` exclu. Les éléments entre 0 et `k-1` sont triés.

**c)** Calculer les distances euclidiennes entre les points de `data` et `pt_search`. Trier par ordre croissant les distances.

**e)** Définir `n2` le nombre de lignes de `data2`. Écrire une boucle `for` en faisant varier `i` entre 0 inclus et `n2` exclu. Appliquer la fonction `algoknn`.

**g)** Définir `n1` le nombre de lignes de `data1` et `n2` le nombre de lignes de `data2`. Écrire une boucle `for` en faisant varier `k` entre 1 inclus et `n1` exclu puis, pour chaque valeur de `k`, écrire une boucle `for` en faisant varier `i` entre 0 inclus et `n2` exclu. Appliquer la fonction `algoknn`.

### 14.2

**a)** Définir `m` le nombre de lignes du fichier. Écrire une boucle `for` en faisant varier `i` entre 1 inclus et `m` exclu.

**b)** Écrire une boucle `for` en faisant varier `i` entre 0 inclus et `len(ptA)` exclu.

**c)** Écrire une boucle `while` tant que l'on n'a pas créé `K` clusters.

**d)** Initialiser une variable `flag_stable` à `False`. Écrire une boucle `while` tant que `flag_stable` vaut `False`. La variable `flag_stable` passe à `True` si un client change de cluster.

### 14.3

**a)** Fonction `init` : liste avec 9 cases « . ». Fonction `choixjoueur` : compter le nombre de cases noires et blanches. Fonction `gain` : ne pas oublier les diagonales dans les tests.

**b)** Effectuer une copie profonde de jeu. Appeler la fonction `choixjoueur`.

**c)** Fonction `valmax` : tester si `listecoups(jeu)` est vide ou si `gain(jeu)!=0`. Sinon parcourir tous les coups possibles de `listecoups(jeu)` et appeler la fonction `valMin`.

**d)** Écrire une boucle `while gain(jeu)==0 and len(listecoups(jeu))!=0`.

# Corrigés des exercices

### 14.1

**a)**

```
def lect_data(fichier):
    # la fonction retourne une liste de listes
    # la liste contient n lignes ; chaque ligne est une liste de 3 éléments
    # n = nombre de lignes du fichier
    f=open(fichier, 'r')      # ouverture de fichier (str) en lecture
    f_données=f.readlines()   # récupère toutes les lignes du fichier
                              # dans la liste f_données
    n=len(f_données)          # nombre de lignes du fichier
    data=[[0 for j in range(3)] for i in range(n)]
            # matrice : n lignes et 3 colonnes
            # taille, gueule et numéro de l'espèce
    for i in range(0, n):     # i varie entre 0 inclus et n exclu
        ligne=f_données[i].strip().split(";")
```

```
        # split(';') permet de séparer la ligne en une liste
        # de mots avec le séparateur ';'
        # strip() permet d'enlever le saut de ligne
        data[i][0], data[i][1]=float(ligne[0]), float(ligne[1])
        if (ligne[2]=="alligator"):
            data[i][2]=0     # numéro 0 désigne alligator
        else:
            data[i][2]=1     # numéro 1 désigne crocodile
    f.close()
    return data
```

**b)**

```
def calc_dist(ptA, ptB):
    # distance euclidienne entre ptA et ptB
    # ptA est une liste de deux éléments
    # ptB est une liste de deux éléments
    return ((ptB[0]-ptA[0])**2+(ptB[1]-ptA[1])**2)**(0.5)
```

**Remarque**

On peut envisager plusieurs définitions de la distance entre $A(x_A, y_A)$ et $B(x_B, y_B)$ :

- Distance euclidienne : $\text{distance} = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$
- Distance de Manhattan : $\text{distance} = |x_B - x_A| + |y_B - y_A|$
- Distance de Tchebychev : $\text{distance} = \max(|x_B - x_A|, |y_B - y_A|)$

**c)** Pour une valeur de *k* fixée, on calcule la distance d'un point à tous les points du fichier de données. On cherche l'espèce majoritaire parmi les *k* plus proches voisins.

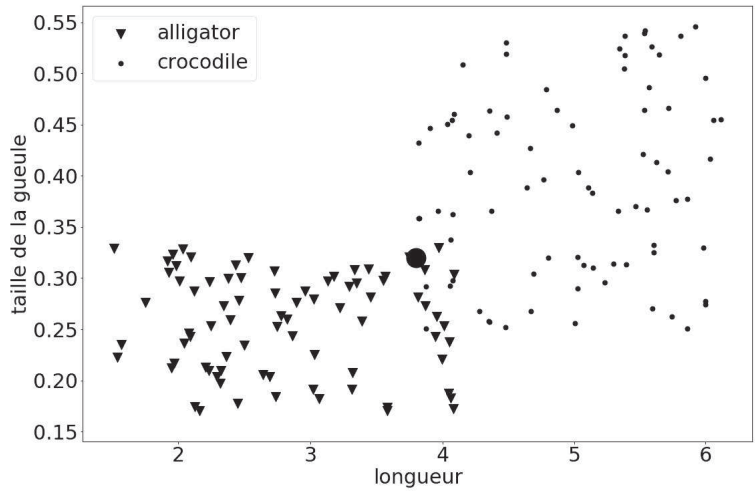
```
def algoknn(data, pt_search, k):
    # calcul de la distance de pt_search à tous les points de data
    # la fonction retourne une prédiction de l'espèce
    # algorithme des k(int) plus proches voisins
    # pt_search : liste de deux valeurs (taille et gueule)
    n=len(data)                    # nombre de lignes de data
    mat_dist=[[0 for j in range(2)] for i in range(n)]
             # matrice avec distance et numéro espèce
    for i in range(len(data)):     # i varie entre 0 inclus et len(data) exclu
        iris1=[data[i][0],data[i][1]]
        mat_dist[i][0]=calc_dist(pt_search,iris1)
        mat_dist[i][1]=data[i][2]
            # on récupère la distance et le nom de l'espèce
    mat_dist.sort()                # tri par ordre croissant des distances
    # on récupère la liste des distances triées par ordre croissant
    # choix de l'espère
    list_nb_espece=[[0, i] for i in range(nb_espece)]
        # liste de listes : (total, numéro de l'espèce)
    for i in range(k):             # i varie entre 0 inclus et k exclu
```

```
            indice=int(mat_dist[i][1]) # mat_dist[i][1] : numéro de l'espèce
            list_nb_espece[indice][0]+=1
    list_nb_espece.sort() # tri de list_nb_espece par ordre croissant de total

    predict=list_nb_espece[nb_espece-1][1]  # le dernier élément est celui
                                            # qui apparaît le plus
    return predict
```

**d)**

```
import matplotlib.pyplot as plt    # module matplotlib.pyplot renommé plt
data=lect_data("fic1.csv")
nb_espece=2                 # on a deux espèces : alligator (0), crocodile (1)
pt_search=[3.8, 0.32]       # taille, gueule
k=5                         # nombre de k plus proches voisins
x1, y1, x2, y2=[],[],[],[]
for i in range(len(data)):
    if data[i][2]==0:
        x1.append(data[i][0])
        y1.append(data[i][1])
    else:
        x2.append(data[i][0])
        y2.append(data[i][1])
plt.figure()
plt.xlabel("longueur")
plt.ylabel("taille de la gueule")
plt.scatter(x1, y1, color='blue', marker='v')
plt.scatter(x2, y2, color='red', marker='.')
plt.scatter(pt_search[0], pt_search[1], color='black', linewidth=8)
plt.legend(['alligator', 'crocodile'])
plt.show()
```

On obtient le graphique suivant avec le fichier « fic1.csv » (voir Site Dunod pour télécharger le fichier de données) :



On applique l'algorithme des $k$ plus proches voisins à `pt_search= [3.8, 0.32]` avec $k = 5$.

Le programme Python renvoie : 0 alligator. L'espèce recherchée (gros cercle noir sur le graphique ci-dessus) est plus près des alligators que des crocodiles.

**Remarque**

L'algorithme des $k$ plus proches voisins est basé sur un algorithme d'apprentissage à partir d'observations étiquetées. Le modèle prédictif est utilisé dans plusieurs cas :

- Régression : le résultat est un réel. Le résultat est la moyenne des valeurs des $k$ plus proches voisins. Exemple : prédiction de la solubilité d'une molécule dans l'eau en mg/mL.
- Classification : le résultat obtenu est une classe d'appartenance (0, 1, 2, …, $C$–1) si on considère $C$ classes possibles. Dans l'exemple précédent, on a $C = 2$ classes (0 pour alligator et 1 pour crocodile).
- Classification binaire : le résultat obtenu est 0 ou 1. Exemples : l'e-mail reçu est-il un spam ? La photo est-elle celle d'un chat ? On utilise un nombre impair de voisins pour ne pas avoir d'*ex aequo*.

**e)**

```
def evalKNN (data1, data2, k):
    # data1 = liste d'apprentissage et data2 = liste de test
    # la fonction retourne la matrice mat
    n2=len(data2)          # nombre de données de la liste de test
    mat=[[0 for j in range(3)] for i in range(3)]
    for i in range(n2):    # i varie entre 0 inclus et n2 exclu
        predict=algoknn(data1, [data2[i][0],data2[i][1]], k)
        mat[int(data2[i][2])][predict]+=1
    return mat
```

**f)** On a plusieurs façons d'évaluer les données de test :

- $Rappel = \frac{TP}{TP+FN}$ = proportion des espèces bien prédites (crocodile) parmi toutes les espèces crocodile dans le fichier de test. Le rappel définit la capacité du modèle à détecter crocodile parmi les espèces crocodile dans le jeu de test. On l'appelle également sensibilité.
- $Spécificité = \frac{TN}{TN+FP}$ = proportion des espèces bien prédites (non crocodile) parmi les espèces qui ne sont pas crocodile dans le fichier de test. La spécificité définit la capacité du modèle à détecter les espèces qui ne sont pas crocodile parmi les espèces qui ne sont pas crocodile dans le jeu de test.
- $Précision = \frac{TP}{TP+FP}$ = proportion des espèces bien prédites (crocodile) parmi toutes les espèces dans le fichier de test. La précision définit la capacité du modèle à détecter crocodile parmi toutes les espèces dans le jeu de test.

**True Positives (TP)**

| | | **Classe réelle** | | |
|---|---|---|---|---|
| | | alligator | crocodile | … |
| **Classe prédite** | alligator | … | … | … |
| | crocodile | … | … | … |
| | … | … | … | … |

**True Negatives (TN)**

| | | **Classe réelle** | | |
|---|---|---|---|---|
| | | alligator | crocodile | … |
| **Classe prédite** | alligator | … | … | … |
| | crocodile | … | … | … |
| | … | … | … | … |

**False Negatives (FN)**

| | | **Classe réelle** | | |
|---|---|---|---|---|
| | | alligator | crocodile | … |
| **Classe prédite** | alligator | … | … | … |
| | crocodile | … | … | … |
| | … | … | … | … |

**False Positives (FP)**

| | | **Classe réelle** | | |
|---|---|---|---|---|
| | | alligator | crocodile | … |
| **Classe prédite** | alligator | … | … | … |
| | crocodile | … | … | … |
| | … | … | … | … |

```
def matconf_indicateurs (data1, data2):
    # data1 = liste d'apprentissage et data2 = liste de test
    # la fonction retourne la matrice de confusion pour crocodile,
    # rappel (float), spécificité(float) et précision(float)
    k=2             # k=2 pour l'algorithme des k plus proches voisins
    mat=evalKNN (data1, data2, k)
    matconf=[[0 for j in range(2)] for i in range(2)]
    matconf[0][0]=mat[1][1]           # True Positives (TP)
    matconf[1][1]=mat[0][0]           # True Negatives (TN)
    matconf[1][0]=mat[0][1]           # False Negatives (FN)
    matconf[0][1]=mat[1][0]           # False Positives (FP)
    TP=matconf[0][0]                  # True Positives (TP)
    FP=matconf[0][1]                  # False Positives (FP)
    FN=matconf[1][0]                  # False Negatives (FN)
    TN=matconf[1][1]                  # True Negatives (TN)
    rappel=TP/(TP+FN)
    spécificité=TN/(TN+FP)
    précision=TP/(TP+FP)
    return matconf, rappel, spécificité, précision
```

**g)** Pour chaque donnée du fichier de test, on exécute l'algorithme des *k* plus proches voisins et on calcule le pourcentage d'espèces mal prédites. On peut ainsi choisir la valeur de *k* permettant d'avoir le plus faible pourcentage d'espèces mal prédites.

```
def predict_k(data1, data2):
    # data1 : liste d'apprentissage
    # data2 : liste de test
    # la fonction retourne la valeur optimale de k
    n1=len(data1)             # nombre de données de la liste d'apprentissage
    n2=len(data2)             # nombre de données de la liste de test
    tab_pred=[[0 for j in range(2)] for i in range(n1)]
    for k in range(1, n1):    # k varie entre 1 inclus et n1 exclu
        nb_erreur=0
        for i in range(n2):   # i varie entre 0 inclus et n2 exclu
            predict=algoknn(data1, [data2[i][0],data2[i][1]], k)
            if predict!=data2[i][2]:
                nb_erreur+=1
        tab_pred[k][0]=nb_erreur
        tab_pred[k][1]=k
    tab_pred.sort()
    return tab_pred,int(tab_pred[1][1])
```

Le programme Python affiche : « Valeur optimisée pour $k = 7$ ».

## 14.2

**a)**

```
def fic_data(fichier):
    # la fonction retourne une liste de listes
    # la liste contient n lignes chaque ligne est une liste de 6 éléments
    # m = nombre de lignes du fichier et n=m-1
    f=open(fichier, 'r')       # ouverture de fichier (str) en lecture
    f_données=f.readlines()    # récupère toutes les lignes du fichier
                               # dans la liste f_données
    m=len(f_données)           # nombre de lignes du fichier
    data=[[0 for j in range(6)] for i in range(m-1)]
    for i in range(1, m):      # i varie entre 1 inclus et m exclu
        ligne=f_données[i].strip().split(",")
        # split(';') permet de séparer la ligne en une liste
        # de mots avec le séparateur ';'
        # strip() permet d'enlever le saut de ligne
        for j in range(6):
            data[i-1][j]=float(ligne[j+2])
    f.close()
    return data
```

**b)**

```
def distance(ptA, ptB):        # distance euclidienne entre ptA et ptB
    som=0
    for i in range(len(ptA)):
        som=som+(ptB[i]-ptA[i])**2
    return som**(0.5)
```

**Remarque**

On peut envisager plusieurs définitions de la distance entre $A(x_A, y_A)$ et $B(x_B, y_B)$ :

- Distance euclidienne : $\text{distance} = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$
- Distance de Manhattan : $\text{distance} = |x_B - x_A| + |y_B - y_A|$
- Distance de Tchebychev : $\text{distance} = \max(|x_B - x_A|, |y_B - y_A|)$

**c)** On choisit aléatoirement $K$ points différents parmi les clients. Il ne faut pas utiliser la boucle `for` avec $K$ étapes puisqu'on peut obtenir deux indices identiques avec la fonction `rd.randint(0, n-1)`.

```
def initcluster(K, data):
    # la fonction retourne une liste de listes contenant les coordonnées
    # des centres des K clusters tirés aléatoirement
    # arguments d'entrée : entier K et matrice data
    import random as rd        # module random renommé rd
    n=len(data)
    nb_col=len(data[0])
    L=[]                # liste de listes : coordonnées des centres des clusters
    liste_indice=[]
    while len(L)!=K:
        i=rd.randint(0, n-1)   # indice aléatoire compris entre
                               # 0 inclus et n-1 inclus
        if i not in liste_indice:
            liste_indice.append(i)
            coordonnées=[]
            for j in range(nb_col):
                coordonnées.append(data[i][j])
            L.append(coordonnées)
    return L                   # liste des K clusters
```

**Remarque**

Les points choisis aléatoirement dans cette étape d'initialisation doivent appartenir au fichier de clients.

**d)**

```
def algokm (K, data):
    # la fonction retourne une liste de listes contenant les coordonnées
    # des centres des K clusters avec l'algorithme des k-moyennes
    # A : liste d'affectation des points i à un cluster
    # arguments d'entrée : entier K = nombre de clusters,
    # data : matrice de données
    # initialisation des K centres des clusters
    L=initcluster(K, data)
    n=len(data)
    nb_col=len(data[0])
```

```
    A=[0 for i in range(n)]   # liste pour affecter les points à un cluster
    flag_stable=False  # flag à False si on affecte un client à un autre cluster
    while flag_stable==False:
        flag_stable=True   # flag initialisé à True
                           # il passe à False si le client i change de cluster
        # on place chaque client dans le cluster k le plus proche
        for i in range(n): # i varie entre 0 inclus et n exclu
            val_min=inf    # initialisation de val_min à infini
            ind_min=0      # indice du cluster correspondant au minimum
            for k in range(K):   # k varie entre 0 inclus et K exclu
                if distance(data[i], L[k])<val_min:
                    val_min=distance(data[i],L[k])
                    ind_min=k
            if A[i]!=ind_min:
                A[i]=ind_min
                flag_stable=False       # le client i a changé de cluster
        if flag_stable==False or K==1:  # teste si on affecte un client
                                        # à un autre cluster
            # on recalcule les coordonnées de chaque cluster
            som_cluster=[[0 for j in range(nb_col)] for i in range(K)]
            nb_el=[0 for i in range(K)]
            ind_compacité=0
            for i in range(n):
                k=A[i]     # le client i appartient au cluster k
                for j in range(nb_col):
                    som_cluster[k][j]+=data[i][j]
                nb_el[k]+=1       # on ajoute un client de plus à ce cluster
                ind_compacité+=distance(data[i], L[k])**2
            for k in range(K):
                if nb_el[k]==0:   # teste si aucun client dans le cluster k
                    coordonnées=[]
                    for j in range(nb_col):
                        coordonnées.append(0)
                else:
                    coordonnées=[]
                    for j in range(nb_col):
                        coordonnées.append(som_cluster[k][j]/nb_el[k])
                L[k]=coordonnées
    return L, ind_compacité, A
```

Cet algorithme a plusieurs défauts :

- Il faut fixer à l'avance la valeur du nombre total de clusters *K*.
- Le résultat dépend fortement du choix des centres initiaux.
- On n'obtient pas nécessairement le résultat optimal.
- On peut obtenir un minimum local qui dépend des centres initiaux.

**Remarque**

Les nouveaux clusters n'appartiennent plus nécessairement au jeu de données comme dans l'étape d'initialisation. On affecte à chaque client le cluster le plus proche. On met à jour les coordonnées des clusters. Ces opérations sont réitérées tant que les clients changent de cluster.

En pratique, on utilise cet algorithme pour former des groupes (ou clusters) inconnus à l'avance mais qu'il faut interpréter ensuite.

**e)**

```
inf=1e10                    # variable représentant l'infini
data=fic_data("clients_data.csv")
n=len(data)
nb_col=len(data[0])

liste_ind_compacité=[]
max_K=6
for K in range(1, max_K):  # K varie entre 1 inclus et max_K exclu
  L,ind_compacité,A=algokm(K, data)
  liste_ind_compacité.append(ind_compacité)
plt.figure()
plt.plot([k for k in range(1, max_K)], liste_ind_compacité)
plt.xlabel('Nombre de clusters')
plt.ylabel('Indicateur de compacité')
plt.show()                  # affiche la figure à l'écran

K=3                         # valeur optimale du nombre total de clusters
L, ind_compacité, A=algokm(K, data)
data_cluster=[[0 for j in range(nb_col)] for i in range(K)]
nbval_cluster=[0 for i in range(K)]
for i in range(n):
    indice=A[i]
    nbval_cluster[indice]+=1
    for j in range(nb_col):
        data_cluster[indice][j]+=data[i][j]

moycluster=[0 for i in range(K)]
for k in range(K):
    som=0
    for i in range(len(data_cluster[0])):
        som+=data_cluster[k][i]
    moycluster[k]=som/nbval_cluster[k]
print("Moyenne des ventes par cluster :", moycluster)
print("Nombre de clients par cluster :", nbval_cluster)
```

**f)** La courbe a l'allure d'un bras. Le nombre optimal de clusters est le point représentant le coude : $K = 3$ (méthode *Elbow*, coude en anglais). L'ajout d'un quatrième cluster n'apporte rien dans la structuration des données. On n'observe presque plus d'amélioration pour l'indicateur de compacité.

Le cluster n° 3 donne les meilleures ventes (68957) avec la majorité des ventes allouées à l'épicerie (27037).

Le cluster n° 2 donne les moins bonnes ventes (22689) avec la majorité des ventes allouées au frais (8296).

Pour une stratégie marketing consistant à faire dépenser davantage les clients, l'entreprise va déployer une campagne auprès des 53 clients du cluster n° 3 en ciblant les catégories surgelé et charcuterie.

**g)**

- Apprentissage supervisé : On cherche une fonction de prédiction à partir d'une base d'apprentissage. Exemple : régression linéaire, algorithme des *k* plus proches voisins.
- Apprentissage non supervisé : Les données ne sont pas étiquetées. Il faut découvrir des groupes. Le programme regroupe les données par groupes, ou clusters (application au regroupement de clients ayant un comportement similaire afin d'adapter les annonces pour le marketing). Exemple : algorithme des *k*-moyennes.

**h)**

```
def predict(L,pt_search):
    # cette fonction retourne l'indice du numéro du cluster le plus
    # proche de pt_search
    # arguments d'entrée : L = liste des coordonnées des clusters
    # L[i] est une liste contenant les coordonnées du cluster d'indice i
    # pt_search : liste contenant les coordonnées de pt_search
    dist_cluster=inf          # distance du pt_search à un centre du cluster
    ind_cluster=0             # indice du numéro du cluster
    for k in range(len(L)):   # k varie entre 0 inclus et len(L) exclu
        if distance(L[k], pt_search)<dist_cluster:
            dist_cluster=distance(L[k],pt_search)
            ind_cluster=k
    return ind_cluster ,L[ind_cluster]

pt_search=[2000, 150, 180, 20, 90, 32]     # exemple pour un client
print(predict(L,pt_search))
print(A[0],pt_search)
```

## 14.3

a)

```
def init():           # début de partie
                      # la fonction retourne une liste jeu avec un plateau vide
                      # les cases vides sont représentées par '.'
    L=9*['.']         # 9 cases avec '.'
    return L

def affiche(jeu):
                      # la fonction affiche le plateau de jeu
                      # jeu=liste de 9 éléments
    print(jeu[0:3])   # 1re ligne du plateau de jeu
    print(jeu[3:6])   # 2e ligne du plateau de jeu
    print(jeu[6:9])   # 3e ligne du plateau de jeu
```

```
def choixjoueur(jeu):
    # la fonction retourne le joueur 'O' ou 'X' devant jouer
    # pour la liste jeu
    som1=0            # initialisation du nombre de cases noires
    som2=0            # initialisation du nombre de cases blanches
    for i in range(9):
        if jeu[i]=='X':
            som1=som1+1   # nombre de cases noires
        elif jeu[i]=='O':
            som2=som2+1   # nombre de cases blanches
    if som1==som2:        # si autant de pions noirs que de pions blancs
        return 'X'        # c'est à X de jouer
    else:
        return 'O'        # c'est à O de jouer

def listecoups(jeu):
    # retourne la liste L des indices des cases vides pour la liste jeu
    L=[]
    for i in range (9):
        if jeu[i]=='.':
            L.append(i)   # ajoute les cases sans pion
    return L

def gain(jeu):
    # retourne 1 si les noirs ont gagné, -1 si les blancs ont gagné
    # et 0 sinon pour la liste jeu
    liste1=['X', 'X', 'X']
    liste2=['O', 'O', 'O']
    if jeu[0:3]==liste1 or jeu[3:6]==liste1 or jeu[6:9]==liste1\
        or [jeu[0], jeu[3], jeu[6]]==liste1\
        or [jeu[1], jeu[4], jeu[7]]==liste1\
        or [jeu[2], jeu[5], jeu[8]]==liste1\
        or [jeu[0], jeu[4], jeu[8]]==liste1\
        or [jeu[2], jeu[4], jeu[6]]==liste1:
        return 1          # 3 pions noirs alignés
    elif jeu[0:3]==liste2 or jeu[3:6]==liste2 or jeu[6:9]==liste2\
        or [jeu[0], jeu[3], jeu[6]]==liste2\
        or [jeu[1], jeu[4], jeu[7]]==liste2\
        or [jeu[2], jeu[5], jeu[8]]==liste2\
        or [jeu[0], jeu[4], jeu[8]]==liste2\
        or [jeu[2], jeu[4], jeu[6]]==liste2:
        return -1         # 3 pions blancs alignés
    else:
        return 0
```

**b)** On ne peut pas écrire `jeu2=jeu` pour réaliser une copie de la liste `jeu`. Si on modifie un élément de `jeu`, alors `jeu2` aura la même modification puisque les deux listes `jeu` et `jeu2` font référence à la même adresse mémoire. On utilise le module `copy` (voir chapitre 1 « Type, listes, loucles, fonctions ») pour réaliser une copie superficielle de `jeu`.

```
def jouercoup(jeu, coup):
    # retourne une nouvelle liste jeu2 en ajoutant un pion
    # à l'indice coup de la liste jeu
    import copy           # module copy
    jeu2=copy.copy(jeu)   # copie superficielle de jeu
                          # si on modifie jeu2, la liste jeu est inchangée
    pion=choixjoueur(jeu)
    if coup in listecoups(jeu):
        jeu2[coup]=pion
    return jeu2
```

**c)**

```
def valMax(jeu):
    # retourne la valeur du maximum du gain et l'indice de la case à jouer
    # pour la liste jeu
    L=listecoups(jeu)
    if len(L)==0 or gain(jeu)!=0:
        # condition d'arrêt de la fonction récursive
        # partie finie car plus de coups à jouer
        # un des joueurs a gagné avec trois pions alignés
        return gain(jeu), -1       # la partie est finie
    else:
        calculmax=-2               # maximum de tous les coups possibles
        ind_coup=-1                # initialisation de l'indice du coup à jouer
        for coup in L:             # parcourt tous les coups possibles
            jeu2=jouercoup(jeu,coup)
            calcul, indice= valMin(jeu2)
            if calcul>calculmax:
                calculmax=calcul   # maximum de tous les coups possibles
                ind_coup=coup      # indice du coup à jouer
        return calculmax, ind_coup # maximum du coup à jouer

def valMin(jeu):
    # retourne la valeur du minimum du gain et l'indice de la case à jouer
    # pour la liste jeu
    L=listecoups(jeu)
    if len(L)==0 or gain(jeu)!=0:
        # condition d'arrêt de la fonction récursive
        # partie finie car plus de coups à jouer
        # un des joueurs a gagné avec trois pions alignés
        return gain(jeu), -1       # la partie est finie
    else:
        calculmin=2                # minimum de tous les coups possibles
```

```
        ind_coup=-1                # initialisation de l'indice du coup à jouer
        for coup in L:             # parcourt tous les coups possibles
            jeu2=jouercoup(jeu,coup)
            calcul, indice=valMax(jeu2)
            if calcul<calculmin:
                calculmin=calcul   # minimum de tous les coups possibles
                ind_coup=coup      # indice du coup à jouer
        return calculmin, ind_coup # minimum du coup à jouer
```

**d)**

```
def jouerminmax(jeu):
    # cette fonction affiche les plateaux de jeu à chaque étape
    # la liste jeu est modifiée à chaque étape
    while gain(jeu)==0 and len(listecoups(jeu))!=0:
        # la partie n'est pas terminée et il reste des coups à jouer
        nomjoueur=choixjoueur(jeu)
        if nomjoueur=='X':        # les noirs jouent
            calcul, ind_coup=valMax(jeu)
            # récupère l'indice du coup à jouer pour les noirs
            jeu=jouercoup(jeu, ind_coup)
            # on passe à l'étape suivante du jeu
        else:
            calcul, ind_coup=valMin(jeu)
            # récupère l'indice du coup à jouer pour les blancs
            jeu=jouercoup(jeu, ind_coup)
            # passe à l'étape suivante du jeu
        print("Joueur qui pose les pions :", nomjoueur)
        affiche(jeu)
    if gain(jeu)==1:
        print("Le joueur X a gagné.")
    elif gain(jeu)==-1:
        print("Le joueur O a gagné.")
    else:
        print("Partie nulle")
```

Le programme principal permettant de visualiser les étapes du jeu de morpion est le suivant :

```
jeu=init()          # début de partie avec cases '.'
jouerminmax(jeu)
```

**e)**

```
def jouercontreIA(jeu):
    # cette fonction affiche les plateaux de jeu à chaque étape
    # la liste jeu est modifiée à chaque étape
    # ordinateur ou IA : pions noirs et l'humain : pions blancs
    while gain(jeu)==0 and len(listecoups(jeu))!=0:
        # la partie n'est pas terminée et il reste des coups à jouer
        nomjoueur=choixjoueur(jeu)
```

```
        if nomjoueur=='X':        # les noirs jouent
            calcul, ind_coup=valMax(jeu)
            # on récupère l'indice du coup à jouer
            jeu=jouercoup(jeu, ind_coup)
            # passe à l'étape suivante du jeu
            print("Joueur qui pose les pions :", nomjoueur)
            affiche(jeu)
        else:
            ind_coup=int(input("Tapez l'indice de la case : "))
            # on récupère l'indice du coup à jouer
            jeu=jouercoup(jeu, ind_coup)
            # passe à l'étape suivante du jeu

    if gain(jeu)==1:
        print("L'ordinateur (joueur X) a gagné.")
    elif gain(jeu)==-1:
        print("Vous avez gagné (joueur O).")
    else:
        print("Partie nulle")
```

Le programme principal permettant de jouer contre l'ordinateur est le suivant :

```
jeu=init()          # début de partie avec cases '.'
jouercontreIA(jeu)
```

**f)**

```
def valMax2(jeu, profondeur):
    # retourne la valeur du maximum du gain et l'indice de la case à jouer
    # pour la liste jeu
    # profondeurmaxi = nombre de coups calculés à l'avance par l'algorithme
    L=listecoups(jeu)
    if len(L)==0 or gain(jeu)!=0 or profondeur ==0:
        # condition d'arrêt de la fonction récursive
        # {partie finie car plus de coups à jouer}
        # ou {un des joueurs a gagné avec trois pions alignés}
        # ou {profondeur nulle} on n'explore pas plus en profondeur l'arbre
        return gain(jeu), -1       # la partie est finie
    else:
        calculmax=-2               # maximum de tous les coups possibles
        ind_coup=-1                # initialisation de l'indice du coup à jouer
        for coup in L:             # parcourt tous les coups possibles
            jeu2=jouercoup(jeu, coup)
            calcul, indice= valMin2(jeu2, profondeur-1)
            if calcul>calculmax:
                calculmax=calcul   # maximum de tous les coups possibles
                ind_coup=coup      # indice du coup à jouer
        return calculmax, ind_coup # maximum du coup à jouer
```

```
def valMin2(jeu, profondeur):
    # retourne la valeur du minimum du gain et l'indice de la case à jouer
    # pour la liste jeu
    # profondeurmaxi = nombre de coups calculés à l'avance par l'algorithme
    L=listecoups(jeu)
    if len(L)==0 or gain(jeu)!=0 or profondeur ==0:
        # condition d'arrêt de la fonction récursive
        # {partie finie car plus de coups à jouer}
        # ou {un des joueurs a gagné avec trois pions alignés}
        # ou {profondeur nulle} on n'explore pas plus en profondeur l'arbre
        return gain(jeu), -1       # la partie est finie
    else:
        calculmin=2                # minimum de tous les coups possibles
        ind_coup=-1                # initialisation de l'indice du coup à jouer
        for coup in L:             # parcourt tous les coups possibles
            jeu2=jouercoup(jeu, coup)
            calcul, indice=valMax2(jeu2, profondeur-1)
            if calcul<calculmin:
                calculmin=calcul    # minimum de tous les coups possibles
                ind_coup=coup       # indice du coup à jouer
        return calculmin, ind_coup  # minimum du coup à jouer
```

**g)**

```
def jouercontreIA3(jeu):
    # cette fonction affiche les plateaux de jeu à chaque étape
    # la liste jeu est modifiée à chaque étape
    # ordinateur ou IA : pions noirs et l'humain : pions blancs
    question="Quel niveau de difficulté souhaitez-vous (2:niveau"+ \
             " débutant, 6:intermédiaire, 10:expert) ? "
    rep=input(question)
    profondeurmaxi=int(rep)
    rep=input("Quels pions voulez-vous ? noirs(X) ou blancs (O) : ")
    while gain(jeu)==0 and len(listecoups(jeu))!=0:
        # la partie n'est pas terminée et il reste des coups à jouer
        nomjoueur=choixjoueur(jeu)
        if nomjoueur!=rep:       # l'ordinateur joue
            if nomjoueur=="X":   # les noirs jouent
                calcul, ind_coup=valMax2(jeu, profondeurmaxi)
            else :               # les blancs jouent
                calcul, ind_coup=valMin2(jeu, profondeurmaxi)
            # récupère l'indice du coup à jouer
            jeu=jouercoup(jeu, ind_coup)
            # passe à l'étape suivante du jeu
            affiche(jeu)
        else:
            ind_coup=int(input("Tapez l'indice de la case : "))
            # récupère l'indice du coup à jouer
```

```
            jeu=jouercoup(jeu, ind_coup)
            # passe à l'étape suivante du jeu

    if gain(jeu)==1:
        print("Le joueur X a gagné.")
    elif gain(jeu)==-1:
        print("Le joueur O a gagné.")
    else:
        print("Partie nulle")
```

Le programme principal permettant de jouer contre l'ordinateur en choisissant le niveau de difficulté et la couleur des pions est :

```
jeu=init()          # début de partie avec cases '.'
jouercontreIA3(jeu)
```

# Base de données (spé)

CHAPITRE 15

Plan



## Thèmes abordés dans les exercices

- Requête `SELECT`
- Clé primaire, clé étrangère
- Opérateurs de comparaison
- Opérateurs ensemblistes `UNION`, `INTERSECT` et `EXCEPT`
- Jointures internes, autojointure
- Agrégation avec les fonctions `MIN`, `MAX`, `SUM`, `AVG` et `COUNT`
- Utilisation de `GROUP BY`

## Points essentiels du cours pour la résolution des exercices

- Identifier une clé primaire et une clé étrangère
- Modèle entité-association
- Compter le nombre d'éléments après avoir regroupé des lignes
- Renommer une colonne, une table
- Utilisation des mots-clés `DISTINCT`, `LIMIT`, `OFFSET`, `ORDER BY`
- Différence entre `WHERE` et `HAVING`

# Les méthodes à retenir

## Modèle entité-association

Une **entité** est un objet, une personne, un concept identifiable dans le cadre d'une base de données. Pour identifier de façon certaine et unique chaque entité, il faut disposer d'une **clé primaire**.

Une **entité** contient plusieurs **attributs** qui prennent des valeurs dans des **domaines** spécifiés à l'avance (le numéro d'une commande est un entier, le total d'une commande est un réel…).

Une commande particulière sera une **instance** du type d'entité `commandes`. On dit également instance de l'entité `commandes`. Un fournisseur particulier sera une instance de l'entité `fournisseurs`.

On distingue deux types d'entités : entité forte (entité qui n'a pas besoin d'une autre entité pour exister) et entité faible (entité qui a besoin d'une autre entité pour pouvoir être définie).

Une **association est un lien entre plusieurs entités**, par exemple entre deux entités (`commandes` et `fournisseurs`). On crée ainsi une association qui relie un fournisseur à une commande.

La **cardinalité** d'une association est constituée de deux entiers représentant le nombre minimal et le nombre maximal d'instances d'une entité en lien avec une instance de l'autre entité.

# Modèle relationnel

Le modèle entité-association est remplacé par un modèle relationnel :

- Chaque entité devient une **relation** (ou **table**). Chaque **attribut** (ou **colonne**) de l'entité devient un attribut de la relation. La clé primaire est souvent le premier attribut de la relation.
- Une association de cardinalité 0..* 0..1 ne donne pas lieu à une nouvelle relation. Il suffit d'ajouter une colonne dans la relation principale contenant la clé primaire de la relation secondaire.
- Chaque association complexe va nécessiter la création d'une nouvelle relation, dont la clé primaire sera le couple des clés primaires des deux entités reliées par l'association.

Une **relation** (ou **table**) est définie par son nom et une liste **d'attributs** (ou **colonnes**) prenant des valeurs dans des domaines spécifiés à l'avance. L'ordre des colonnes n'est pas significatif.

Une **instance** de la relation est une **ligne** (ou **enregistrement**) de la table. Un tel élément est également appelé **tuple** de la relation. L'ordre des lignes n'est pas significatif. Il n'y a pas de duplication de lignes.

La base de données `GESTION` est constituée de deux tables `commandes` et `fournisseurs`, dont les attributs respectifs sont définis ci-dessous.

La table `commandes` contient les colonnes :

- `num_comm` : clé primaire, de type entier, désigne le numéro de commande ;
- `annee` : de type entier, désigne l'année où est payée la commande ;
- `mois` : de type entier, désigne le mois où est payée la commande ;
- `num_fourn` : de type entier, désigne le numéro d'identification du fournisseur ;
- `total`, de type flottant, désigne le prix total de la commande.

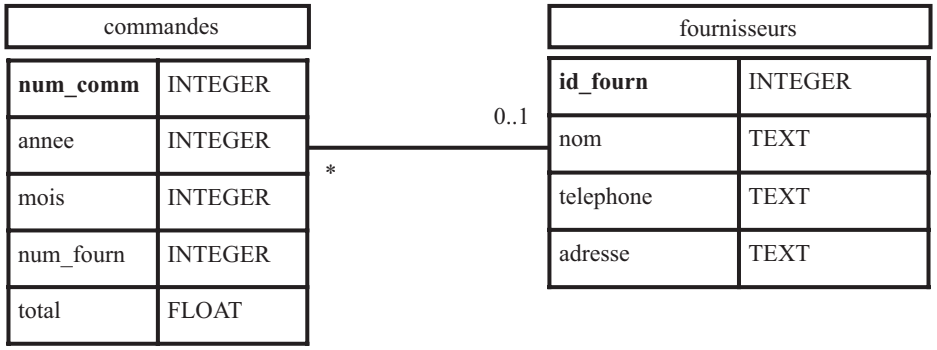
| num_comm | annee | mois | num_fourn | total |
|---|---|---|---|---|
| 1 | 2020 | 2 | 2 | 250 |
| 2 | 2019 | 1 | 1 | 400 |
| 3 | 2020 | 12 | 2 | 300 |
| 4 | 2018 | 5 | 2 | 420 |
| 5 | 2020 | 6 | 3 | 420 |

La table `fournisseurs` contient les colonnes :

- `id_fourn` : clé primaire, de type entier, désigne le numéro d'identification du fournisseur ;
- `nom` : de type chaîne de caractères, désigne le nom du fournisseur ;
- `telephone` : de type chaîne de caractères, désigne le numéro de téléphone du fournisseur ;
- `adresse` : de type chaîne de caractères, désigne l'adresse du fournisseur.

| id_fourn | nom | telephone | adresse |
|---|---|---|---|
| 1 | Lectra | 05 57 97 80 00 | Cestas |
| 2 | Pierron | 03 87 95 14 77 | Sarreguemines |
| 3 | Jeulin | 08 25 56 35 63 | Evreux |
| 4 | Electrome | 05 56 39 69 18 | Mérignac |

Un schéma relationnel est constitué d'un nom et d'une liste d'attributs, avec un type de données pour chaque attribut. Le schéma de la base de données est l'ensemble des schémas de relation :




Une **clé primaire (*primary key* ou PK)** sert à identifier une ligne de manière unique. Chaque ligne de la table `commandes` est désignée par un numéro d'identifiant unique `num_comm`. Chaque fournisseur est désigné par un numéro d'identifiant unique `id_fourn`. Les clés primaires sont souvent représentées en gras ou soulignées.

**Remarque**

Une clé primaire n'est pas forcément associée à un unique attribut (voir exercice 15.3 « Analyse d'un 100 mètres ») même si c'est le cas le plus fréquent.

Une commande peut être liée à aucun fournisseur (commande en préparation) ou à un seul fournisseur. On écrit alors la multiplicité 0..1.

Un fournisseur peut être lié à aucune commande, une ou plusieurs commandes (sans aucune limite). On écrit alors la multiplicité 0..* que l'on note *.

| Multiplicité | Abréviation | Cardinalité |
|---|---|---|
| 0..0 | 0 | Aucune ligne |
| 0..1 | | Aucune ou une seule ligne |
| 1..1 | 1 | Une seule ligne |
| 0..* | * | Aucune, une ou plusieurs lignes (pas de limite) |
| 1..* | | Au moins une ligne (pas de limite) |
| x..x | x | Exactement x ligne(s) |
| m..n | | Au moins m et au plus n lignes |

Une association est un lien entre deux ou plusieurs entités.

L'association est de type **1 – * (un à plusieurs)**. Cette association possède 1 comme cardinalité maximale pour la table `fournisseurs` et une cardinalité multiple pour la table `commandes`. Une commande ne peut être liée qu'à un seul fournisseur au maximum (cardinalité 0..1).

Une **clé étrangère (*foreign key* ou FK)** sert à lier des relations (ou tables) entre elles. La clé étrangère `num_fourn` permet d'avoir le numéro d'identifiant du fournisseur. La clé `num_fourn` n'est pas une clé primaire car plusieurs commandes peuvent avoir le même fournisseur.

# Types d'associations

On considère trois cas :

- **Association de type 1 – 1 (un à un)** : la cardinalité maximale vaut 1 pour les deux entités. On ajoute une clé étrangère dans une des deux tables.
- **Association de type 1 – * (un à plusieurs)** : la cardinalité maximale vaut 1 pour une entité et est strictement supérieure à 1 pour l'autre entité. La clé étrangère doit être dans la table correspondant à la cardinalité multiple.
- **Association de type * – * (plusieurs à plusieurs)** : la cardinalité maximale est strictement supérieure à 1 pour les deux tables. Il faut créer une table d'association composée d'au moins deux clés étrangères.

# Création de la base de données `GESTION`

On pourra tester les requêtes SQL en utilisant SQLite Database Browser (https://sqlitebrowser.org). Les requêtes SQL `CREATE TABLE` et `INSERT INTO` ne sont pas au programme des concours. Elles permettent de créer la base de données `GESTION`.

```
CREATE TABLE commandes(num_comm integer PRIMARY KEY, annee integer,
mois integer,num_fourn integer, total real) ;
INSERT INTO commandes VALUES(1,2020,2,2,250) ;
INSERT INTO commandes VALUES(2,2019,1,1,400) ;
INSERT INTO commandes VALUES(3,2020,12,2,300) ;
INSERT INTO commandes VALUES(4,2018,5,2,420) ;
INSERT INTO commandes VALUES(5,2020,6,3,420) ;
CREATE TABLE fournisseurs (id_fourn integer PRIMARY KEY, nom text,
telephone text,adresse text) ;
INSERT INTO fournisseurs VALUES(1,'Lectra','05 57 97 80 00','Cestas') ;
INSERT INTO fournisseurs VALUES(2,'Pierron',
'03 87 95 14 77','Sarreguemines') ;
INSERT INTO fournisseurs VALUES(3,'Jeulin','08 25 56 35 63','Evreux') ;
INSERT INTO fournisseurs VALUES(4,'Electrome','05 56 39 69 18','Mérignac') ;
```

# Requête `SELECT` avec simple clause `WHERE` (sélection)

La requête SQL renvoie toutes les informations de toutes les commandes :

```
SELECT * FROM commandes ;
```

**Remarque**

On peut écrire les requêtes SQL sur plusieurs lignes pour améliorer la lisibilité.

La requête de base pour rechercher des données est la commande `SELECT` :

```
SELECT *
FROM table
WHERE condition ;
```

`*` permet de retourner toutes les colonnes.

On n'utilise pas d'accent ni d'espace pour désigner les colonnes (ou attributs). Le type des colonnes peut être entier (`INTEGER`), réel (`FLOAT`) ou chaîne de caractères (`TEXT`).

La casse (c'est-à-dire minuscule ou majuscule) n'a pas d'importance pour la désignation des objets.

Le caractère point-virgule est un terminateur d'instruction. Il n'est pas obligatoire de l'écrire.

On peut ajouter des opérateurs `AND`, `OR` dans la condition `WHERE`.

```
SELECT *
FROM table
WHERE condition1 AND condition2 ;
```

Cette requête SQL renvoie les numéros de commande et le total des commandes dont le total est strictement supérieur à 200 et dont l'année est strictement inférieure à 2020 :

```
SELECT num_comm, total
FROM commandes
WHERE total>200 AND annee<2020 ;
```

Cette requête SQL renvoie les numéros de commande dont le total est strictement inférieur à 300 et dont l'année est supérieure ou égale à 2020 :

```
SELECT num_comm
FROM commandes
WHERE total < 300 AND annee >=2020 ;
```

Les opérateurs de comparaison sont :

| > | < | >= | <= | = | <> |
|---|---|---|---|---|---|
| strictement supérieur à | strictement inférieur à | supérieur ou égal à | inférieur ou égal à | égal à | différent de |

**Remarque**

On utilise la même syntaxe avec Python pour l'opérateur de comparaison « différent de ».

# Projection

La projection de la **relation** `R` suivant les **attributs** `A1, A2,…,Ai` extrait de la table `R` les colonnes correspondant aux attributs spécifiés. On la note $\Pi_{A_1,\ldots A_i}(R)$.

L'opération $\Pi_{num\_comm,total}(commandes)$ donne la table suivante :

| num_comm | **total** |
|---|---|
| 1 | 250 |
| 2 | 400 |
| 3 | 300 |
| 4 | 420 |
| 5 | 420 |

La requête SQL s'écrit :

```
SELECT num_comm, total
FROM commandes ;
```

## Renommage AS

`AS` permet de renommer une colonne ou une table.

```
SELECT table.colonne1 AS nouvellecolonne1, colonne2 AS nouvellecolonne2
FROM table AS t ;
```

On peut omettre `AS` :

```
FROM table t
```

On peut écrire `t.colonne1` au lieu de `table.colonne1`.

On peut écrire `FROM table t` au lieu de `FROM table AS t`.

## Mots-clés `DISTINCT`, `LIMIT`, `OFFSET`, `ORDER BY`

On peut ajouter `DISTINCT` après `SELECT` pour éviter d'afficher des lignes en double.

```
SELECT DISTINCT colonne1
FROM table ;
```

La commande `LIMIT 3` permet de sélectionner les trois premiers résultats.

```
SELECT colonne1, colonne2
FROM table
LIMIT 3 ;
```

La commande `LIMIT 3 OFFSET 2` permet de sélectionner les trois premiers résultats sans utiliser les deux premiers de `table`.

```
SELECT colonne1, colonne2
FROM table
LIMIT 3 OFFSET 2 ;
```

La commande `ORDER BY total ASC` permet de trier par ordre croissant, où `total` est le champ sur lequel s'effectue le tri.

```
SELECT total, annee FROM commandes ORDER BY total ASC ;
```

Par défaut, le tri est ascendant. On peut écrire `ORDER BY total` sans ajouter `ASC`.

La commande `ORDER BY total DESC` permet de trier par ordre décroissant, où `total` est le champ sur lequel s'effectue le tri.

```
SELECT total, annee FROM commandes ORDER BY total DESC ;
```

## Opérateurs ensemblistes `UNION`, `INTERSECT` et `EXCEPT`

On crée la table `fournisseurs2`.

```
CREATE TABLE fournisseurs2 (id_fourn integer PRIMARY KEY, nom text,
telephone text,adresse text) ;
INSERT INTO fournisseurs2 VALUES(1,'Radiometer','01 49 44 35 50','Neuilly') ;
INSERT INTO fournisseurs2 VALUES(2,'Objectif Bastille',
'01 43 43 57 38','Paris') ;
INSERT INTO fournisseurs2 VALUES(3,'Photo Denfert','01 43 35 14 92','Paris') ;
```

| id_fourn | nom | telephone | adresse |
|---|---|---|---|
| 1 | Radiometer | 01 49 44 35 50 | Neuilly |
| 2 | Objectif Bastille | 01 43 43 57 38 | Paris |
| 3 | Photo Denfert | 01 43 35 14 92 | Paris |

La commande `UNION` permet d'obtenir la réunion des enregistrements de deux requêtes `SELECT`. Pour chaque requête `SELECT`, on doit avoir le même nombre de colonnes et le même type pour chaque colonne. Les enregistrements identiques sont affichés une seule fois.

```
SELECT nom, adresse FROM fournisseurs
UNION
SELECT nom, adresse FROM fournisseurs2
ORDER BY nom ;
```

La commande `INTERSECT` permet d'obtenir l'intersection de deux requêtes `SELECT`, c'est-à-dire les enregistrements communs aux deux requêtes (voir exercice 15.6 « Numéros de sécurité sociale »).

La commande `EXCEPT` permet de récupérer les enregistrements de la première requête `SELECT` sans inclure les résultats de la deuxième requête `SELECT` (voir exercice 15.6 « Numéros de sécurité sociale »).

## Jointure, produit cartésien

Pour mettre en relation deux tables, on peut utiliser `JOIN… ON…` qui réalise une jointure.

Les jointures permettent de mettre en relation plusieurs tables. Dans la plupart des cas, on impose l'égalité des valeurs d'une colonne d'une table à celles d'une colonne d'une autre table.

```
SELECT *
FROM commandes
JOIN fournisseurs ;
```

Cette requête réalise le produit cartésien des deux tables `commandes` et `fournisseurs`. À chaque ligne de la table `commandes`, il accole l'ensemble des lignes de la table `fournisseurs`. Le nombre de lignes affichées vaut 5×4=20.

On a réalisé une jointure entre les deux tables.

| num_comm | annee | mois | num_fourn | total | id_fourn | nom | telephone | adresse |
|---|---|---|---|---|---|---|---|---|
| 1 | 2020 | 2 | 2 | 250.0 | 1 | Lectra | 05 57 97 80 00 | Cestas |
| 1 | 2020 | 2 | 2 | 250.0 | 2 | Pierron | 03 87 95 14 77 | Sarreguemines |
| 1 | 2020 | 2 | 2 | 250.0 | 3 | Jeulin | 08 25 56 35 63 | Evreux |
| 1 | 2020 | 2 | 2 | 250.0 | 4 | Electrome | 05 56 39 69 18 | Mérignac |
| 2 | 2019 | 1 | 1 | 400.0 | 1 | Lectra | 05 57 97 80 00 | Cestas |

On peut préciser une condition de jointure avec le mot-clé `ON`.

```
SELECT *
FROM commandes
JOIN fournisseurs ON id_fourn=num_fourn ;
```

La requête SQL n'affiche pas les 20 lignes mais uniquement celles dont la condition `id_fourn=num_four` est vérifiée. On a réalisé une jointure interne. On aurait pu écrire `INNER JOIN` au lieu de `JOIN`.

Une **autojointure** consiste à joindre une table à elle-même. On renomme les deux tables pour éviter toute confusion (voir exercice 15.6 « Numéros de sécurité sociale »).

## Agrégation avec les fonctions `MIN`, `MAX`, `SUM`, `AVG` et `COUNT`. Utilisation de `GROUP BY`

Les fonctions d'agrégation `SUM(e)`, `AVG(e)`, `MAX(e)`, `MIN(e)`, `COUNT(e)`, `COUNT(*)` calculent respectivement la somme, la moyenne arithmétique, le maximum, le minium, le nombre de valeurs non nulles de l'expression `e` et le nombre de lignes pour chaque groupe de lignes défini par la clause `GROUP BY`. Si la requête ne comporte pas de clause `GROUP BY`, le calcul est effectué pour l'ensemble des lignes sélectionnées par la requête.

La fonction `max(total)` renvoie le maximum de toutes les commandes :

```
SELECT max(total)
FROM commandes ;
```

On obtient : 420.

**Remarque**

Il ne faut pas confondre, d'une part, `AVG` pour les requêtes SQL et, d'autre part, `np.mean()` de la bibliothèque `numpy` avec Python, qui calculent dans les deux cas la moyenne.

La fonction `COUNT(*)` compte le nombre de lignes :

```
SELECT COUNT(*)
FROM commandes ;
```

On obtient 5.

Dans certains cas, on veut éviter d'avoir plusieurs fois la même ligne. On utilise alors la commande `GROUP BY`, qui permet de regrouper les lignes en une seule. La commande `COUNT(*)` compte alors le nombre de lignes concernées.

```
SELECT *
FROM table
WHERE condition
GROUP BY expression ;
```

La requête SQL renvoie le nombre de commandes ayant le même total :

```
SELECT total, COUNT(*)
FROM commandes
GROUP BY total ;
```

Pour chaque total, on obtient le nombre de commandes ayant ce total :

| total | COUNT(*) |
|---|---|
| 250 | 1 |
| 300 | 1 |
| 400 | 1 |
| 420 | 2 |

La commande `GROUP BY` évite d'avoir plusieurs fois la même ligne. On regroupe les lignes d'un même total en une seule. La commande `COUNT(*)` compte alors le nombre de lignes concernées.

La commande `ORDER BY total ASC` permet de trier par ordre croissant, où `total` est le champ sur lequel s'effectue le tri.

```
SELECT total, COUNT(*)
FROM commandes
GROUP BY total
ORDER BY total ASC ;
```

Par défaut, le tri est ascendant. On peut écrire : `ORDER BY total` sans ajouter `ASC`.

La commande `ORDER BY total DESC` permet de trier par ordre décroissant, où `total` est le champ sur lequel s'effectue le tri.

```
SELECT total, COUNT(*)
FROM commandes
GROUP BY total
ORDER BY total DESC ;
```

Cette requête SQL renvoie le total des commandes passées en 2020 :

```
SELECT SUM(total)
FROM commandes
WHERE annee=2020
GROUP BY année ;
```

La requête SQL renvoie la moyenne des commandes pour chaque année :

```
SELECT annee, AVG(total)
FROM commandes
GROUP BY année ;
```

Cette requête SQL renvoie le nom des fournisseurs et l'adresse pour les commandes passées en 2020 :

```
SELECT nom, adresse
FROM fournisseurs
JOIN commandes ON num_fourn=id_fourn
WHERE annee=2020
GROUP BY nom ;
```

**Remarque**

Si on veut mettre en relation une autre table, il faut écrire une ligne supplémentaire `JOIN… ON…`

On peut écrire également :

```
SELECT nom, adresse
FROM fournisseurs, commandes
WHERE annee=2020 AND num_fourn=id_fourn
GROUP BY nom ;
```

Voici la requête SQL qui renvoie le fournisseur pour lequel le plus de commandes ont été passées en 2018 et les années suivantes :

```
SELECT nom, COUNT(*)
FROM fournisseurs
JOIN commandes ON num_fourn=id_fourn
WHERE annee>=2018
GROUP BY nom
ORDER BY COUNT(*) DESC
LIMIT 1 ;
```

La commande `LIMIT 1` permet d'afficher le premier enregistrement.

## Filtrage des agrégats avec `HAVING`

`HAVING` fait quasiment la même chose que `WHERE` mais permet d'utiliser des fonctions d'agrégation comme `COUNT` pour compter le nombre d'éléments, `AVG` pour récupérer la moyenne, `MAX` pour récupérer le maximum, `MIN` pour récupérer le minimum et `SUM` pour calculer la somme.

La requête est alors la suivante :

```
SELECT *
FROM table1
JOIN table2 ON table1.colonne1=table2.colonne2
GROUP BY expression
HAVING COUNT(*)>=1 ;
```

Dans le cas général, on a :

```
SELECT *
FROM table1
JOIN table2 ON table1.colonne1=table2.colonne2
WHERE condition
GROUP BY expression HAVING COUNT(*)>=1
```

```
ORDER BY colonne1
LIMIT 3 OFFSET 2 ;
```

La requête SQL renvoie la liste des fournisseurs et le nombre de commandes pour lesquels plus de 2 commandes ont été passées pour chaque fournisseur en 2020 :

```
SELECT nom, COUNT(*)
FROM fournisseurs
JOIN commandes ON id_fourn=num_fourn
WHERE annee=2020
GROUP BY nom
HAVING COUNT(*)>=2 ;
```

## Requêtes imbriquées

Une requête imbriquée en SQL peut retourner un champ (ici le maximum de la colonne `total`) ou une colonne.

On cherche à obtenir le nom du fournisseur qui a passé la commande dont le prix total est le plus élevé. En cas d'égalité du prix total, on obtiendra les noms de tous les fournisseurs possibles.

```
SELECT nom
FROM fournisseurs
JOIN commandes ON num_fourn=id_fourn
WHERE total=(SELECT MAX(total) FROM commandes)
GROUP BY nom ;
```

# Énoncés des exercices

## 15.1

**Mesures de houle (Mines Ponts 2018)**

On dispose d'une base de données relationnelle `Vagues`. La première table est `Bouee`. On se limite aux attributs suivants : le numéro d'identification `idBouee`, le nom du site `nomSite`, le nom de la mer ou de l'océan `localisation`, le type du capteur `typeCapteur` et la fréquence d'échantillonnage `frequence`.

| idBouee | nomSite | localisation | typeCapteur | frequence |
|---|---|---|---|---|
| 831 | Porquerolles | Méditerranée | Datawell non directionnelle | 2.00 |
| 291 | Les pierres noires | Mer d'iroise | Datawell directionnelle | 1.28 |
| … | … | … | … | … |

La deuxième table est `Campagne`.

On se limite aux attributs suivants : le numéro d'identification `idCampagne`, le numéro d'identification de la bouée `idBouee`, la date de début `debutCampagne` et la date de fin `finCampagne`.

| idCampagne | idBouee | debutCampagne | finCampagne |
|---|---|---|---|
| 08301 | 831 | 01/01/2010 00h00 | 15/01/2010 00h00 |
| 02911 | 291 | 15/10/2005 18h30 | 18/10/2005 08h00 |
| … | … | … | … |

La troisième table est `Tempete`. Les informations fournies relatives à un événement « tempête » sont les suivantes :

- date de début et fin de tempête,
- valeur maximale de hauteur de vague $H_{max}$,
- le détail de certains paramètres non définis ici, obtenus au pic de tempête.

On se limite aux attributs suivants : le numéro d'identification de la tempête `idTempete`, le numéro d'identification de la bouée `idBouee`, la date de début `debutTempete`, la date de fin `finTempete`, la valeur maximale de hauteur de vague `Hmax`.

| idTempete | idBouee | debutTempete | finTempete | Hmax |
|---|---|---|---|---|
| 083010 | 831 | 07/01/2010 20h00 | 09/01/2010 15h30 | 5.3 |
| … | … | … | … | … |

Le schéma de la base de données est donc : `Vagues = {Bouee, Campagne, Tempete}`.

Formuler les requêtes SQL permettant de répondre aux questions suivantes :

**a)** Quels sont le numéro d'identification et le nom de site des bouées localisées en Méditerranée ?

**b)** Quel est le numéro d'identification des bouées où il n'y a pas eu de tempêtes ?

**c)** Pour chaque site, quelle est la hauteur maximale enregistrée lors d'une tempête ?

## 15.2

■■□□

**Autour des nombres premiers (Mines Ponts 2019)**

Au cours du développement des fonctions nécessaires à la manipulation des nombres premiers, on s'aperçoit que le choix des algorithmes pour évaluer chaque fonction est primordial pour garantir des performances acceptables. On souhaite donc mener des tests à grande échelle pour évaluer les performances réelles du code qui a été développé. Pour ce faire, on effectue un grand nombre de tests sur une multitude d'ordinateurs. Les données sont ensuite centralisées dans une base de données composée de deux tables.

La première table est `ordinateurs` et permet de stocker des informations sur les ordinateurs utilisés pour les tests. Ses attributs sont :

- `nom TEXT` : clé primaire, le nom de l'ordinateur ;
- `gflops INTEGER` : la puissance de l'ordinateur en milliards d'opérations flottantes par seconde ;
- `ram INTEGER` : la quantité de mémoire vive de l'ordinateur en Go.

Exemple du contenu de cette table :

| nom | gflops | ram |
|---|---|---|
| ---------- | ---------- | ---------- |
| nyarlathotep165 | 69 | 32 |
| nyarlathotep145 | 137 | 32 |
| … | | |
| shubniggurath28 | 133 | 16 |
| azathoth145 | 85 | 8 |

La seconde table est `fonctions` et stocke les informations sur les tests effectués pour différentes fonctions en cours de développement. Ses attributs sont :

- `id INTEGER` : l'identifiant du test effectué ;
- `nom TEXT` : le nom de la fonction testée (par exemple li, Ei, etc.) ;
- `algorithme TEXT` : le nom de l'algorithme qui permet le calcul de la fonction testée (par exemple BBS si on teste une fonction de génération de nombres aléatoires) ;
- `teste_sur TEXT` : le nom du PC sur lequel le test a été effectué ;
- `temps_exec INTEGER` : le temps d'exécution du test en millisecondes.

Exemple du contenu de cette table :

| id | nom | algorithme | teste_sur | temps_exec |
|---|---|---|---|---|
| ---------- | ---------- | ---------- | ---------- | ---------- |
| 1 | li | rectangles | nyarlathotep165 | 2638 |
| 2 | li | rectangles | shubniggurath28 | 736 |
| 3 | li | trapezes | nyarlathotep165 | 4842 |
| … | | | | |
| 2154 | Ei | Puiseux | nyarlathotep145 | 2766 |
| 2155 | aleatoire | BBS | azathoth145 | 524 |

**a)** Expliquer pourquoi il n'est pas possible d'utiliser l'attribut `nom` comme clé primaire de la table `fonctions`.

**b)** Écrire une requête SQL permettant de connaître le nombre d'ordinateurs disponibles et leur quantité moyenne de mémoire vive.

**c)** Écrire une requête SQL permettant d'extraire les noms des PC sur lesquels l'algorithme `rectangles` n'a pas été testé pour la fonction nommée `li`.

**d)** Écrire une requête SQL permettant, pour la fonction nommée `Ei`, de trier les résultats des tests du plus lent au plus rapide. Pour chaque test, retenir le nom de l'algorithme utilisé, le nom du PC sur lequel il a été effectué et la puissance du PC.

## 15.3

■□□□

**Analyse d'un 100 mètres (Banque PT 2018)**

Le 100 m est une épreuve d'athlétisme consistant à courir sur une distance de 100 m en ligne droite en une durée la plus faible possible. Une analyse des besoins des entraîneurs a conduit à la définition d'une base de données constituée de trois tables, dont les attributs respectifs sont partiellement définis ci-dessous :

| **Table** `coureurs` | | |
|---|---|---|
| **Attribut** | **Type** | **Description** |
| `id` | Entier | Identifiant du coureur (clé primaire) |
| `nom` | Chaîne | Nom du coureur |
| `prenom` | Chaîne | Prénom du coureur |

| **Table** `epreuves` | | |
|---|---|---|
| **Attribut** | **Type** | **Description** |
| `id` | Entier | Identifiant de l'épreuve (clé primaire) |
| `nom` | Chaîne | Nom précis de l'épreuve |
| `date` | Chaîne | Date de l'épreuve au format 'AAAA-MM-JJ' |
| `distance` | Flottant | Distance parcourue (m) |

| Table `performances` | | |
|---|---|---|
| **Attribut** | **Type** | **Description** |
| `id_coureur` | Entier | Identifiant du coureur |
| `id_epreuve` | Entier | Identifiant de l'épreuve |
| `temps` | Flottant | Temps, ou instant d'arrivée (s) |
| `inst_cte` | Flottant | Instant initial de la phase à vitesse constante (s) |
| `inst_dec` | Flottant | Instant initial de la phase de décélération (s) |
| `travail` | Flottant | Travail massique de la force de propulsion (J/kg) |

On précise que, pour la table `performances`, la clé primaire est constituée du couple formé par les deux identifiants `id_coureur` et `id_epreuve`.

**a)** Justifier que ce couple d'identifiants constitue bien une clé primaire pour cette table.

**b)** Écrire une requête SQL renvoyant les noms et dates de toutes les épreuves de 100 m enregistrées dans la base.

**c)** Que fait la requête suivante ?

```
SELECT p.temps, p.inst_cte, p.inst_dec, p.travail
FROM performances p
JOIN epreuves e ON e.id = p.id_epreuve
WHERE e.distance = 100 AND temps <=12 ;
```

**d)** Écrire une requête SQL renvoyant les noms et prénoms des coureurs, les noms et dates des épreuves, et les temps réalisés pour tous les 100 m enregistrés dans la base.

## 15.4

■■□□

### Capacité d'une route (CCP PSI 2017)

Il existe plusieurs formes de congestions routières, selon leur cause : la congestion récurrente, la congestion « prévisible » (travaux, manifestations, météo) et la congestion due aux incidents et accidents, par définition imprévisibles. On s'intéresse ici au niveau de congestion récurrente qui peut être défini comme le surplus de demande qui amène la congestion. Pour réaliser les mesures, la chaussée est équipée de boucles électromagnétiques connectées aux stations qui remontent l'information vers un système central. Ces boucles permettent de compter le nombre de véhicules qui passent sur les routes et, dans le cadre de cette étude, il s'agit de boucles doubles qui permettent aussi d'estimer les vitesses. La méthodologie choisie ici pour estimer la capacité d'une route est celle utilisée notamment par les centres d'études techniques de l'équipement. Elle consiste à représenter un diagramme fondamental, c'est-à-dire le débit $q$, nombre de véhicules par unité de temps, en fonction de la concentration $c$, nombre de véhicules par unité de longueur. Ce diagramme permet alors de déterminer les paramètres caractéristiques de la route.

L'ensemble des données produites par le réseau est archivé dans une base de données. Les variables d'intérêt moyennées par tranche de temps de 6 min pour chaque point de mesure sont le débit `q_exp`, représentatif du nombre de véhicules par unité de temps et la vitesse moyenne `v_exp` des véhicules en ce point. On peut également accéder à la concentration, `c_exp`, par calcul étant donné que $\text{c\_exp} = \dfrac{\text{q\_exp}}{\text{v\_exp}}$.

Une version simplifiée de cette base de données est réduite à deux tables.

La table `STATIONS` répertorie les stations de mesures ; elle contient les attributs :

- `id_station` (clé primaire) : entier identifiant chaque station ;
- `nom` : chaîne de caractères désignant le nom de la station ;
- `nombre_voies` : entier donnant le nombre de voies de la section d'autoroute.

La table `COMPTAGES` répertorie les différents enregistrements de données réalisés au cours du temps par les stations de comptage. Elle contient les attributs :

- `id_comptage` : entier identifiant chaque comptage ;
- `id_station` : entier identifiant la station concernée ;
- `date` : entier datant la mesure ;
- `voie` : entier numérotant la voie sur laquelle a été effectuée la mesure ;
- `q_exp` : flottant donnant le débit mesuré pendant 6 minutes ;
- `v_exp` : flottant donnant la vitesse moyenne mesurée pendant 6 minutes.

**a)** L'étude se focalise uniquement sur les mesures de l'une des stations, la M8B. Écrire une requête SQL qui renvoie les données de comptage (`id_comptage`, `date`, `voie`, `q_exp`, `v_exp`) mesurées à la station de comptage de nom M8B.

Le résultat de la requête précédente est stocké dans une nouvelle table `COMPTAGES_M8B` à cinq colonnes (`id_comptage`, `date`, `voie`, `q_exp`, `v_exp`).

On fait l'hypothèse que les mesures sur les différentes voies d'une même station sont enregistrées de façon synchronisée. Lors d'un enregistrement pour une station à trois voies, on écrit donc trois lignes dans la table `COMPTAGES` avec trois dates identiques. Pour chacun des enregistrements de la station M8B, trois lignes avec trois dates identiques sont donc présentes dans la nouvelle table `COMPTAGES_M8B`. Pour la suite de l'étude, les résultats expérimentaux de chacune des trois voies doivent être agrégés pour se ramener à une voie unique.

**b)** Écrire une requête SQL qui renvoie, pour chaque date des données de `COMPTAGES_M8B`, le débit correspondant à la somme des débits de chaque voie.

**c)** De la même façon, une requête SQL permet d'obtenir la moyenne des vitesses sur l'ensemble des trois voies pour chaque date des données de `COMPTAGES_M8B`. Écrire cette requête SQL.

## 15.5

■■□□

**Réseau routier (Mines Ponts 2017)**

On modélise un réseau routier par un ensemble de *croisements* et de *voies* reliant ces croisements. Les voies partent d'un croisement et arrivent à un autre croisement. Ainsi, pour modéliser une route à double sens, on utilise deux voies circulant en sens opposés.

La base de données du réseau routier est constituée des tables suivantes :

- `Croisement (id, longitude, latitude)`
- `Voie (id, longueur, id_croisement_debut, id_croisement_fin)`

Dans la suite on considère *c* l'identifiant (`id`) d'un croisement donné.

**a)** Écrire la requête SQL qui renvoie les identifiants des croisements atteignables en utilisant une seule voie à partir du croisement ayant l'identifiant *c*.

**b)** Écrire la requête qui renvoie les longitudes et latitudes des croisements atteignables en utilisant une seule voie, à partir du croisement *c*.

**c)** Que renvoie la requête suivante ?

```
SELECT V2.id_croisement_fin
FROM Voie as V1
JOIN Voie as V2
ON V1.id_croisement_fin=V2.id_croisement_debut
WHERE V1.id_croisement_debut=c ;
```

## 15.6

■■□□

### Numéros de sécurité sociale

On considère la base de données `SECURITE_SOCIALE` pour gérer les numéros de sécurité sociale des assurés. Pour chaque région, on a une table qui contient les attributs :

- `id_personne` : de type entier, identifie chaque personne ;
- `nom` : de type chaîne de caractères, désigne le nom de la personne ;
- `prénom` : de type chaîne de caractères, désigne le prénom de la personne ;
- `annee` : de type entier, désigne l'année de naissance ;
- `numsecu` : de type flottant, désigne le numéro de sécurité sociale ;
- `id_personne2` : de type entier, identifie le mari ou la femme ;
- `rembour` : de type flottant, désigne la somme à rembourser à `id_personne`.

`tab_bretagne`

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | DURAND | Alfred | 1980 | 1801223255521 | 3 | 50.5 |
| 2 | DUPONT | Thomas | 1985 | 1851056568848 | | 100.8 |
| 3 | MAUREL | Juliette | 1985 | 1750950504544 | 1 | 350 |
| 4 | DJOKOVIC | Anne | 1970 | 1701584545321 | | |
| 5 | BERDYCH | Bertrand | 1989 | 2892502545458 | 6 | 10 |
| 6 | CHEMIN | Marie | 1989 | 2825084584848 | 5 | 18 |

`tab_aquitaine`

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | BOULEAU | Patrick | 1975 | 1755258458181 | 2 | 100 |
| 2 | BOULEAU | Marie | 1978 | 2508584545451 | 1 | 80 |
| 3 | CHASSAT | Paul | 1980 | 1803355484812 | | 18 |
| 4 | FALQUIER | Anne | 1985 | 2885884788882 | 5 | 400.5 |
| 5 | DUPE | Bertrand | 1986 | 1865254848482 | 4 | 500 |
| 6 | CHEMIN | Marie | 1983 | 2825358874851 | | 29 |

**a)** Écrire une requête SQL qui renvoie le nom, le prénom, le numéro de sécurité sociale et l'année de naissance des assurés dont l'année de naissance est supérieure ou égale à 1980 pour la Bretagne et 1985 pour la Nouvelle-Aquitaine. Les noms des assurés sont affichés par ordre croissant.

**b)** Écrire une requête qui renvoie la liste des couples de Bretagne. Chaque ligne contiendra le nom et le prénom de l'assuré ainsi que le nom et le prénom de son mari ou de sa femme.

**c)** Écrire une requête qui renvoie le nom et le prénom des assurés qui ont les mêmes noms et prénoms en Bretagne et Nouvelle-Aquitaine.

**d)** Écrire une requête qui renvoie le nom et le prénom des assurés de Bretagne sauf ceux qui ont même nom et même prénom en Bretagne et Nouvelle-Aquitaine.

**e)** Écrire une requête qui renvoie les deux plus grands remboursements des couples de Nouvelle-Aquitaine. Chaque ligne contiendra le nom, le prénom et le remboursement.

**f)** Écrire une requête qui renvoie la moyenne des remboursements par année de naissance des assurés de Bretagne. Le minimum du remboursement des assurés pour chaque année de naissance doit être supérieur ou égal à 50. Les années sont affichées par ordre décroissant.

# Du mal à démarrer ?

## 15.1

**a)** Utiliser `SELECT`, `FROM` et `WHERE`.

**b)** Utiliser `NOT IN (SELECT…)`.

**c)** Utiliser `MAX(Hmax)` et `GROUP BY`.

## 15.2

**a)** Plusieurs lignes ont le même nom pour la table `fonctions`.

**b)** Utiliser `COUNT` et `AVG`.

**c)** Utiliser `AND` et l'opérateur de comparaison <>.

**d)** Utiliser `JOIN… ON…`

## 15.3

**a)** Un coureur court une seule fois un 100 m.

**b)** Utiliser `SELECT`, `FROM` et `WHERE`.

**d)** Utiliser `JOIN… ON…`

## 15.4

**a)** Utiliser `JOIN… ON…`

**b)** Utiliser `SUM(q_exp)` avec `GROUP BY date`

**c)** Utiliser `AVG(v_exp)`

## 15.5

**a)** Utiliser `SELECT… FROM… WHERE`.

**b)** Utiliser `SELECT… FROM Croisement JOIN Voie ON WHERE`.

## 15.6

**a)** Utiliser `SELECT… FROM… UNION… SELECT FROM…`

**b)** Utiliser `SELECT… FROM tab_bregagne AS t1 JOIN tab_bretagne AS t2 ON…`

**c)** Utiliser `SELECT… FROM… INTERSECT… SELECT…`

**d)** Utiliser `SELECT… FROM… EXCEPT… SELECT…`

**f)** Utiliser `SELECT… FROM… GROUP BY… HAVING MIN…`

# Corrigés des exercices

## 15.1

Les requêtes SQL `CREATE TABLE` et `INSERT INTO` ne sont pas au programme des concours. Elles permettent de créer la base de données de l'exercice et de tester les réponses des questions de l'exercice.

```
CREATE TABLE Bouee(idBouee INTEGER PRIMARY KEY,nomSite TEXT,
    localisation TEXT, typeCapteur TEXT, frequence FLOAT) ;
INSERT INTO Bouee VALUES (831, 'Porquerolles', 'Méditerranée',
    'Datawell non directionnelle',2.00) ;
INSERT INTO Bouee VALUES (291, 'Les pierres noires',
    "Mer d'iroise",'Datawell directionnelle',1.28) ;
INSERT INTO Bouee VALUES (851, 'Arcachon', 'Atlantique',
    'Datawell non directionnelle',2.00) ;
CREATE TABLE Campagne(idCampagne INTEGER PRIMARY KEY,idBouee INTEGER,
    debutCampagne TEXT, finCampagne TEXT) ;
INSERT INTO Campagne VALUES (08301,831, '01/01/2010 00h00',
    '15/01/2010 00h00') ;
INSERT INTO Campagne VALUES (02911,291, '15/10/2005 18h30',
    '18/10/2015 08h00') ;
CREATE TABLE Tempete(idTempete INTEGER PRIMARY KEY,idBouee INTEGER,
    debutTempete TEXT, finTempete TEXT,Hmax FLOAT) ;
INSERT INTO Tempete VALUES (083011,831, '07/01/2010 20h00',
    '09/01/2010 15h30',5.3) ;
INSERT INTO Tempete VALUES (083012,831, '10/01/2010 20h00',
    '11/01/2010 15h30',6.3) ;
INSERT INTO Tempete VALUES (029012,291, '16/10/2005 08h30',
    '18/10/2005 09h00',8.5) ;
```

**a)**

```
SELECT idBouee, nomSite
FROM Bouee
WHERE localisation='Méditerranée' ;
```

Si on exécute la requête SQL, on obtient : 831, Porquerolles.

**b)**

```
SELECT Bouee.idBouee
FROM Bouee
WHERE Bouee.idBouee NOT IN (SELECT Tempete.idBouee FROM Tempete) ;
```

Si on exécute la requête SQL, on obtient : 851.

**c)**

```
SELECT nomSite,MAX(Hmax)
FROM Bouee
JOIN Tempete ON Bouee.idBouee=Tempete.idBouee
GROUP BY Bouee.idBouee ;
```

Si on exécute la requête SQL, on obtient :

| nomSite | MAX(Hmax) |
|---|---|
| Les pierres noires | 8.5 |
| Porquerolles | 6.3 |

## 15.2

Les requêtes SQL `CREATE TABLE` et `INSERT INTO` ne sont pas au programme des concours. Elles permettent de créer la base de données de l'exercice et de tester les réponses des questions de l'exercice.

```
CREATE TABLE ordinateurs(nom text, gflops integer, ram integer) ;
INSERT INTO ordinateurs VALUES('nyarlathotep165',69,32) ;
INSERT INTO ordinateurs VALUES('nyarlathotep145',137,32) ;
INSERT INTO ordinateurs VALUES('shubniggurath28',133,16) ;
INSERT INTO ordinateurs VALUES('azathoth145',85,8) ;
CREATE TABLE fonctions(id integer PRIMARY KEY, nom text, algorithme text,
    teste_sur text, temps_exec integer) ;
INSERT INTO fonctions VALUES(1,'li','rectangles','nyarlathotep165',2638) ;
INSERT INTO fonctions VALUES(2,'li','rectangles','shubniggurath28',736) ;
INSERT INTO fonctions VALUES(3,'li','trapezes','nyarlathotep165',4842) ;
INSERT INTO fonctions VALUES(4,'Ei','Puiseux','nyarlathotep145',2766) ;
INSERT INTO fonctions VALUES(5,'aleatoire','BBS','azathoth145',524) ;
```

**a)** Une clé primaire sert à identifier une ligne de manière unique. L'attribut `nom` ne permet pas d'identifier une ligne de manière unique puisque les lignes 1, 2 et 3 ont le même nom.

**b)**

```
SELECT COUNT(*), AVG(ram)
FROM ordinateurs ;
```

**c)**

```
SELECT teste_sur
FROM fonctions
WHERE fonctions.nom='li' AND algorithme <>'rectangles' ;
```

**d)**

```
SELECT fonctions.algorithme, ordinateurs.nom,gflops
FROM fonctions
JOIN ordinateurs ON teste_sur=ordinateurs.nom
WHERE fonctions.nom='Ei'
ORDER BY temps_exec ;
```

Autre possibilité :

```
SELECT fonctions.algorithme, ordinateurs.nom,gflops
FROM fonctions,ordinateurs
WHERE teste_sur=ordinateurs.nom AND fonctions.nom='Ei'
ORDER BY temps_exec ;
```

## 15.3

Les requêtes SQL `CREATE TABLE` et `INSERT INTO` ne sont pas au programme des concours. Elles permettent de créer la base de données de l'exercice et de tester les réponses des questions de l'exercice.

```
CREATE TABLE coureurs(id INTEGER, nom TEXT, prenom TEXT) ;
INSERT INTO coureurs VALUES (1, 'coureur1', 'Prénom1') ;
INSERT INTO coureurs VALUES (2, 'coureur2', 'Prénom2') ;
INSERT INTO coureurs VALUES (3, 'coureur3', 'Prénom3') ;


CREATE TABLE epreuves(id INTEGER, nom TEXT, date TEXT, distance FLOAT) ;
INSERT INTO epreuves VALUES (1, 'épreuve1', '2018-08-01', 100) ;
INSERT INTO epreuves VALUES (2, 'épreuve2', '2018-07-01', 100) ;
INSERT INTO epreuves VALUES (3, 'épreuve3', '2018-07-01', 115.5) ;
INSERT INTO epreuves VALUES (4, 'épreuve4', '2018-07-01', 120.5) ;


CREATE TABLE performances(id_coureur INTEGER, id_epreuve INTEGER,
    temps FLOAT, inst_cte FLOAT, inst_dec FLOAT, travail FLOAT) ;
INSERT INTO performances VALUES (1, 1, 10.2, 2, 4, 100) ;
INSERT INTO performances VALUES (2,1, 10.5, 2.5, 4.5, 110) ;
INSERT INTO performances VALUES (3,1, 10.8, 2.4, 4.8, 110) ;
INSERT INTO performances VALUES (1, 2, 10.2, 2, 4, 100) ;
INSERT INTO performances VALUES (1, 3, 5, 2, 4, 100) ;
```

**a)** Une clé primaire sert à identifier une ligne de manière unique. Chaque coureur parcourt une seule fois une épreuve. Le couple d'identifiants (`id_coureur` et `id_epreuve`) est donc unique.

**b)**

```
SELECT nom, date
FROM epreuves e
WHERE distance=100 ;
```

On renomme la table `epreuves`. On pourrait écrire :

```
FROM epreuves AS e ;
```

Si on exécute la requête SQL, on obtient :

| nom | date |
|---|---|
| épreuve1 | 2018-08-01 |
| épreuve2 | 2018-07-01 |

**c)** La requête affiche l'instant d'arrivée, l'instant initial de la phase à vitesse constante, l'instant initial de la phase de décélération et le travail massique pour toutes les épreuves de 100 m ayant un instant d'arrivée inférieur ou égal à 12 s.

**d)**

```
SELECT c.nom, prenom, e.nom, e.date, temps
FROM coureurs c
JOIN performances p ON p.id_coureur=c.id
JOIN epreuves e ON e.id=p.id_epreuve
WHERE distance=100 ;
```

Autre possibilité :

```
SELECT c.nom, prenom, e.nom, e.date, temps
FROM coureurs c, performances p, epreuves e
WHERE distance=100 AND p.id_coureur=c.id AND e.id=p.id_epreuve ;
```

| nom | prenom | nom | date | temps |
|---|---|---|---|---|
| coureur1 | Prénom1 | épreuve1 | 2018-08-01 | 10.2 |
| coureur1 | Prénom1 | épreuve2 | 2018-07-01 | 10.2 |
| coureur2 | Prénom2 | épreuve1 | 2018-08-01 | 10.5 |
| coureur3 | Prénom3 | épreuve1 | 2018-08-01 | 10.8 |

On renomme les tables `performances`, `épreuves` et `coureurs`. On pourrait écrire :

```
FROM coureurs AS c
```

## 15.4

**a)**

```
SELECT id_comptage, date, voie, q_exp, v_exp
FROM COMPTAGES
JOIN STATIONS ON STATIONS.id_station = COMPTAGES.id_station
WHERE nom='M8B' ;
```

**b)**

```
SELECT date, SUM(q_exp)
FROM COMPTAGES_M8B
GROUP BY date ;
```

**c)**

```
SELECT date, AVG(v_exp)
FROM COMPTAGES_M8B
GROUP BY date ;
```

## 15.5

**a)**

```
SELECT id_croisement_fin FROM Voie
WHERE id_croisement_debut=c ;
```

**Attention**

Une voie est définie par un croisement de début (`id_croisement_debut`) et un croisement de fin (`id_croisement_fin`). Il n'y a donc pas d'autre croisement entre `id_croisement_debut` et `id_croisement_fin`.

Lorsque les voitures peuvent circuler dans les deux sens entre deux croisements, il faut considérer deux voies dans la table `Voie`.

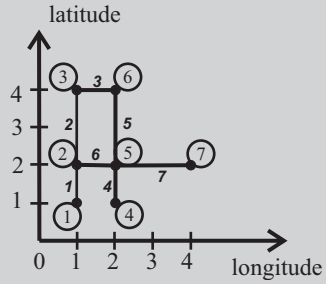
**Remarque**

On peut tester la requête SQL avec les données suivantes :

7 croisements : (1,1,1) ; (2,1,2) ; (3,1,4) ; (4,2,1) ; (5,2,2) ; (6,2,4) ; (7,4,2).

14 voies : (1,1,1,2) ; (2,2,2,3) ; (3,1,3,6) ; (4,1,4,5) ; (5,2,5,6) ; (6,1,2,5) ; (7,2,5,7) ; (8,1,2,1) ; (9,2,3,2) ; (10,1,6,3) ; (11,1,5,4) ; (12,2,6,5) ; (13,1,5,2) ; (14,2,7,5).

Le graphe suivant représente les croisements et les voies. Les identifiants des croisements sont entourés par un cercle. Les identifiants des voies sont indiqués entre deux croisements.




Si on exécute la requête SQL avec $c = 5$, on obtient :

| id |
|---|
| 6 |
| 7 |
| 4 |
| 2 |

On a 4 croisements atteignables en utilisant une seule voie à partir du croisement $c = 5$.

**b)**

```
SELECT longitude, latitude FROM Croisement
JOIN Voie ON Voie.id_croisement_debut=5
WHERE Voie.id_croisement_fin=Croisement.id ;
```

**Remarque**

Si on exécute la requête SQL avec $c = 5$, on obtient :

| longitude | latitude |
|---|---|
| 2 | 4 |
| 4 | 2 |
| 2 | 1 |
| 1 | 2 |

**c)** La requête SQL affiche le croisement de fin d'un trajet comprenant exactement deux voies en partant du croisement ayant l'identifiant $c$.

**Remarque**

Si on exécute la requête SQL avec $c = 5$, on obtient :

| id |
|---|
| 3 |
| 5 |
| 5 |
| 5 |
| 1 |
| 3 |
| 5 |

On peut considérer 7 trajets comprenant deux voies en partant du croisement ayant l'identifiant $c = 5$ : (5-6-3) ; (5-2-5) ; (5-7-5) ; (5-6-5) ; (5-2-1) ; (5-2-3) et (5-4-5).

## 15.6

**a)** La commande `UNION` permet d'obtenir la réunion des enregistrements de deux requêtes `SELECT`. Pour chaque requête `SELECT`, on doit avoir le même nombre de colonnes et le même type pour chaque colonne. Les enregistrements identiques sont affichés une seule fois.

```
SELECT nom, prenom FROM tab_bretagne
UNION
SELECT nom, prenom FROM tab_aquitaine
ORDER BY nom ;
```

On obtient :

| | |
|---|---|
| BERDYCH | Bertrand |
| BOULEAU | Marie |
| BOULEAU | Patrick |
| CHASSAT | Paul |
| CHEMIN | Marie |
| DJOKOVIC | Anne |
| DUPE | Bertrand |
| DUPONT | Thomas |
| DURAND | Alfred |
| FALQUIER | Anne |
| MAUREL | Juliette |

```
SELECT nom, prenom, numsecu, annee FROM tab_bretagne WHERE annee>=1980
UNION
SELECT nom, prenom, numsecu, annee FROM tab_aquitaine WHERE annee>=1985
ORDER BY nom ;
```

**b)** Une autojointure consiste à joindre une table à elle-même. On peut afficher sur une même ligne une personne ou son mari ou sa femme. On renomme les deux tables pour éviter toute confusion.

```
SELECT t1.nom,t1.prenom, t2.nom, t2.prenom
FROM tab_bretagne AS t1
JOIN tab_bretagne AS t2
ON t1.id_personne2=t2.id_personne ;
```

Cette requête affiche des doublons.

On obtient :

| | | | |
|---|---|---|---|
| DURAND | Alfred | MAUREL | Juliette |
| MAUREL | Juliette | DURAND | Alfred |
| BERDYCH | Bertrand | CHEMIN | Marie |
| CHEMIN | Marie | BERDYCH | Bertrand |

```
SELECT t1.nom,t1.prenom, t2.nom, t2.prenom
FROM tab_bretagne AS t1
JOIN tab_bretagne AS t2
ON t1.id_personne2=t2.id_personne
WHERE t1.id_personne2>t1.id_personne ;
```

On obtient :

| | | | |
|---|---|---|---|
| DURAND | Alfred | MAUREL | Juliette |
| BERDYCH | Bertrand | CHEMIN | Marie |

**c)** La commande `INTERSECT` permet d'obtenir l'intersection de deux requêtes `SELECT`, c'est-à-dire les enregistrements communs aux deux requêtes.

```
SELECT nom, prenom FROM tab_bretagne
INTERSECT
SELECT nom, prenom FROM tab_aquitaine ;
```

**d)** La commande `EXCEPT` permet de récupérer les enregistrements de la première requête `SELECT` sans inclure les résultats de la deuxième requête `SELECT`.

```
SELECT nom, prenom  FROM tab_bretagne
EXCEPT
SELECT nom, prenom FROM tab_aquitaine
ORDER BY nom ;
```

On obtient :

| | |
|---|---|
| BERDYCH | Bertrand |
| DJOKOVIC | Anne |
| DUPONT | Thomas |
| DURAND | Alfred |
| MAUREL | Juliette |

**e)**

```
SELECT t1.nom, t1.prenom, t2.nom, t2.prenom, t1.rembour+t2.rembour
FROM tab_aquitaine AS t1
JOIN tab_aquitaine AS t2
ON t1.id_personne2=t2.id_personne
ORDER BY t1.rembour+t2.rembour DESC
LIMIT 2 ;
```

**Remarque**

On peut écrire :

```
SELECT t1.nom, t1.prenom, t2.nom, t2.prenom, t1.rembour+t2.rembour
FROM tab_aquitaine t1
JOIN tab_aquitaine t2
ON t1.id_personne2=t2.id_personne
ORDER BY t1.rembour+t2.rembour DESC
LIMIT 2 ;
```

On obtient :

| | | | | |
|---|---|---|---|---|
| FALQUIER | Anne | DUPE | Bertrand | 900 |
| DUPE | Bertrand | FALQUIER | Anne | 900 |

**f)**

```
SELECT annee, AVG(rembour)
FROM tab_bretagne
GROUP BY annee
HAVING MIN(rembour)>=50
ORDER BY annee DESC ;
```

On obtient :

| | |
|---|---|
| 1985 | 225.0 |
| 1980 | 50.0 |

# Algorithmique numérique (uniquement TSI et TPC)

CHAPITRE 16

Plan



## Thèmes abordés dans les exercices

- Méthode d'Euler
- Équation différentielle du premier ordre
- Équation différentielle du deuxième ordre
- Résolution d'un système linéaire par la méthode de Gauss
- Interpolation polynomiale de Lagrange
- Interpolation par morceaux

## Points essentiels du cours pour la résolution des exercices

- Utilisation de listes
- Identifier les indices de début et de fin dans les boucles `for`
- Transformer une équation différentielle du deuxième ordre en deux équations différentielles du premier ordre
- Écrire un problème sous forme matricielle
- Déterminer le maximum des éléments dans une colonne
- Programmer le calcul du pivot partiel
- Programmer le polynôme d'interpolation de Lagrange

# Les méthodes à retenir

## Méthode d'Euler

On cherche à résoudre numériquement l'équation différentielle : $\left.\frac{\mathrm{d}u}{\mathrm{d}t}\right|_t = F\left(u(t), t\right)$.

La fonction $u(t)$ est une fonction continue du temps (qui est également continu). On réalise la discrétisation du signal en récupérant les valeurs de la tension $u$ à intervalles de temps réguliers $h$. On appelle $h$ le pas de calcul et $N$ le nombre d'échantillons. On obtient deux suites de nombres :

- $\{t_i = ih$, $i$ variant de 0 à $(N-1)\}$,
- $\{u_i = u(t = ih)$, $i$ variant de 0 à $(N-1)\}$.

Avec Python, on définit deux listes `T` et `U` telles que $\begin{cases} T[i] = t_i \\ U[i] = u_i \end{cases}$.

On a alors :

- $\{T[i] = t_i = ih)$, $i$ variant de 0 à $(N-1)\}$,
- $\{U[i] = u_i = u(t = ih)$, $i$ variant de 0 à $(N-1)\}$.

On considère un point d'abscisse $t_i$. On applique la formule de Taylor au premier ordre au voisinage de $t_i$ :

$$u(t_{i+1}) = u(t_i) + (t_{i+1} - t_i)\left(\left(\frac{\mathrm{d}u}{\mathrm{d}t}\right)_{t_i} + \ldots\right.$$

On pose $t_i = ih$ et $t_{i+1} = (i+1)h = t_i + h$. On peut écrire la formule de Taylor sous la forme :

$$u(t_{i+1}) = u(t_i) + h\left(\frac{\mathrm{d}u}{\mathrm{d}t}\right)_{t_i} + \ldots$$

$$\left(\frac{\mathrm{d}u}{\mathrm{d}t}\right)_{t_i} \approx \left(\frac{\Delta u}{\Delta t}\right)_{t_i} = \frac{u(t_{i+1}) - u(t_i)}{h} = \frac{u_{i+1} - u_i}{h} = \frac{\mathrm{U[i+1]} - \mathrm{U[i]}}{h}$$

La méthode d'Euler est une méthode du premier ordre.

- La méthode d'Euler explicite consiste à évaluer $\left(\frac{\mathrm{d}u}{\mathrm{d}t}\right)_{t_i}$ en utilisant la valeur de la dérivée à l'ancienne position, c'est-à-dire à l'instant $t_i$ : $\left(\frac{\mathrm{d}u}{\mathrm{d}t}\right)_{t_i} = F(u_i, t_i)$. On a un schéma explicite puisqu'on calcule chaque point en fonction du point précédent. La relation de récurrence s'écrit : $\frac{u_{i+1} - u_i}{h} = F(u_i, t_i)$.
- La méthode d'Euler implicite consiste à évaluer $\left(\frac{\mathrm{d}u}{\mathrm{d}t}\right)_{t_i}$ en utilisant la valeur de la dérivée à la nouvelle position, c'est-à-dire à l'instant $t_{i+1}$ : $\left(\frac{\mathrm{d}u}{\mathrm{d}t}\right)_{t_i} = F(u_{i+1}, t_{i+1})$. La relation de récurrence s'écrit : $\frac{u_{i+1} - u_i}{h} = F(u_i, t_i)$.

Sauf indication contraire, on utilisera la méthode d'Euler explicite dans les exercices.

### Exemple de résolution avec la méthode d'Euler

On applique la méthode d'Euler explicite pour résoudre l'équation différentielle : $\left.\frac{\mathrm{d}u}{\mathrm{d}t}\right|_t = -3(u(t) - 5)$. On considère $N = 500$ points. Le pas de calcul, noté $h$ vaut 3,3 ms. On suppose que, à $t = 0$, $u(0) = 1$.

La méthode d'Euler explicite permet de donner une évaluation de la dérivée première à l'instant $t_i$ :

$$\left(\frac{\mathrm{d}u}{\mathrm{d}t}\right)_{t_i} \approx \left(\frac{\Delta u}{\Delta t}\right)_{t_i} = \frac{u(t_{i+1}) - u(t_i)}{h}$$

On en déduit une expression approchée de l'équation différentielle avec le schéma explicite :

$$\left(\frac{\mathrm{d}u}{\mathrm{d}t}\right)_{t_i} \approx \frac{u\left(t_{i+1}\right)-u\left(t_i\right)}{h} = -3\left(u\left(t_i\right)-5\right)$$

La relation de récurrence s'écrit : $u\left(t_{i+1}\right) = u\left(t_i\right) + h\left(-3\left(u\left(t_i\right)-5\right)\right)$.

En utilisant les listes `T` et `U`, on en déduit les relations de récurrence :

$$\begin{cases} T\left[i+1\right] = T\left[i\right] + h \\ U\left[i+1\right] = U\left[i\right]\left(1-3h\right) + 15h \end{cases}$$

On construit au fur et à mesure les listes `T` et `U`.

```
N=500                           # nombre de points
u0=1                            # condition initiale
h=3.3e-3                        # pas de calcul
T=[0]
U=[u0]                          # premier élément de la liste U
for i in range(N-1):            # i varie entre 0 inclus et N-1 exclu
    T.append(T[i]+h  )          # calcul de T[i+1]
    U.append(U[i]*(1-3*h)+15*h) # calcul de U[i+1]
```

`T=[0]` définit une liste contenant un seul élément : l'instant initial $t = 0$.

`U=[u0]` définit une liste contenant un seul élément : la valeur initiale de $u$.

La boucle `for` permet d'ajouter au fur et à mesure des éléments dans les listes `T` et `U`.

- À la fin de la première étape de la boucle `for`, les listes `T` et `U` contiennent deux éléments : $\left[0,h\right]$ et $\left[u\left(0\right),u\left(h\right)\right]$.
- À la fin de la deuxième étape de la boucle `for`, on obtient : $\left[0,h,2h\right]$ et $\left[u\left(0\right),u\left(h\right),u\left(2h\right)\right]$.
- À la fin de la dernière étape de la boucle `for`, on obtient : $\left[0,h,2h,...,\left(n-1\right)h\right]$ et $\left[u\left(0\right),u\left(h\right),u\left(2h\right),...,u\left(\left(n-1\right)h\right)\right]$.

Il faut faire varier $i$ entre 0 inclus et $n - 1$ exclu pour obtenir les listes `T` et `U`.

## Système linéaire de Cramer

Un système linéaire est de Cramer d'ordre $n$ si c'est un système de $n$ équations linéaires à $n$ inconnues et s'il admet une solution unique.

On cherche à résoudre le système $PQ = R$. On prend par exemple : $P = \begin{pmatrix} 2 & 4 & 2 \\ 1 & 4 & 2 \\ 2 & 9 & 3 \end{pmatrix}$ ; $Q = \begin{pmatrix} q_0 \\ q_1 \\ q_2 \end{pmatrix}$ et $R = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$.

Le système s'écrit alors : $\begin{pmatrix} 2 & 4 & 2 \\ 1 & 4 & 2 \\ 2 & 9 & 3 \end{pmatrix}\begin{pmatrix} q_0 \\ q_1 \\ q_2 \end{pmatrix} = \begin{pmatrix} 2q_0+4q_1+2q_2 \\ q_0+4q_1+2q_2 \\ 2q_0+9q_1+3q_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$.

Avant de décrire l'algorithme, on pose $P' = P = \begin{pmatrix} 2 & 4 & 2 \\ 1 & 4 & 2 \\ 2 & 9 & 3 \end{pmatrix}$ et $R' = R = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$.

On considère $n$ étapes successives. L'étape $p$ (avec $p$ compris entre 0 et $n-1$) se décompose de la façon suivante :

- Recherche du pivot partiel (élément maximum en valeur absolue dans la colonne $p$) = $P'_{i,p}$ tel que $i \geq p$.
- Permutation des lignes $i$ et $p$ pour $P'$ et $R'$ afin que le pivot soit sur la diagonale.
- Division de la ligne $p$ par le pivot pour $P'$ et $R'$. On obtient alors $P'_{p,p} = 1$.
- Remplacement pour $P'$ et $R'$ des lignes $i \neq p$ par la combinaison linéaire de la ligne $i$ et de la ligne $p$ : $\text{ligne}(i) \leftarrow \text{ligne}(i) - P'_{i,p}\ \text{ligne}(p)$. On obtient alors $P'_{i,p} = 0$ avec $i \neq p$.

**Étape $p = 0$ :**

$P' = \begin{pmatrix} \boxed{2} & 4 & 2 \\ 1 & 4 & 2 \\ 2 & 9 & 3 \end{pmatrix}$. Le pivot est $P'_{i=0,p=0} = 2$ avec $i \geq p$. Les étapes 1 et 2 sont inutiles. On divise la ligne 0 par le pivot.

On obtient : $P' = \begin{pmatrix} 1 & 2 & 1 \\ 1 & 4 & 2 \\ 2 & 9 & 3 \end{pmatrix}$ et $R' = \begin{pmatrix} 0{,}5 \\ 1 \\ 1 \end{pmatrix}$.

Combinaison linéaire $i \neq p$ :

- $i = 1$ : valeur $= P'_{i=1,p=0} = 1$. D'où : $\text{ligne}(i=1) \leftarrow \text{ligne}(i=1) - \text{valeur} \times \text{ligne}(p=0)$. On a alors : $P' = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 2 & 1 \\ 2 & 9 & 3 \end{pmatrix}$ et $R' = \begin{pmatrix} 0{,}5 \\ 0{,}5 \\ 1 \end{pmatrix}$.
- $i = 2$ : valeur $= P'_{i=2,p=0} = 2$. D'où : $\text{ligne}(i=2) \leftarrow \text{ligne}(i=2) - \text{valeur} \times \text{ligne}(p=0)$. On a alors : $P' = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 2 & 1 \\ 0 & 5 & 1 \end{pmatrix}$ et $R' = \begin{pmatrix} 0{,}5 \\ 0{,}5 \\ 0 \end{pmatrix}$.

**Étape $p = 1$ :**

$P' = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 2 & 1 \\ 0 & \boxed{5} & 1 \end{pmatrix}$. Le pivot est $P'_{i=2,p=1} = 5$ avec $i \geq p$. On permute les lignes $i$ et $p$. On obtient alors : $P' = \begin{pmatrix} 1 & 2 & 1 \\ 0 & \boxed{5} & 1 \\ 0 & 2 & 1 \end{pmatrix}$ et $R' = \begin{pmatrix} 0{,}5 \\ 0 \\ 0{,}5 \end{pmatrix}$.

On divise la ligne $p = 1$ par le pivot. On obtient alors $P'_{p=1,p=1} = 1$ et $P' = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 1 & 1/5 \\ 0 & 2 & 1 \end{pmatrix}$ et $R' = \begin{pmatrix} 0{,}5 \\ 0 \\ 0{,}5 \end{pmatrix}$.

Combinaison linéaire $i \neq p$ :

- $i = 0$ : valeur $= P'_{i=0,p=1} = 2$. D'où : $\text{ligne}(i=0) \leftarrow \text{ligne}(i=0) - \text{valeur} \times \text{ligne}(p=1)$. On a alors : $P' = \begin{pmatrix} 1 & 0 & 3/5 \\ 0 & 1 & 1/5 \\ 0 & 2 & 1 \end{pmatrix}$ et $R' = \begin{pmatrix} 0{,}5 \\ 0 \\ 0{,}5 \end{pmatrix}$.

- $i = 2$ : valeur $= P'_{i=2,p=1} = 2$. D'où : ligne$(i=2) \leftarrow$ ligne $(i=2) -$ valeur $\times$ ligne$(p=1)$. On a alors : $P' = \begin{pmatrix} 1 & 0 & 3/5 \\ 0 & 1 & 1/5 \\ 0 & 0 & 3/5 \end{pmatrix}$ et $R' = \begin{pmatrix} 0{,}5 \\ 0 \\ 0{,}5 \end{pmatrix}$.

**Étape $p = 2$ :**

$P' = \begin{pmatrix} 1 & 0 & 3/5 \\ 0 & 1 & 1/5 \\ 0 & 0 & \boxed{3/5} \end{pmatrix}$. Le pivot est $P'_{i=2,p=2} = \frac{3}{5}$ avec $i \geq p = 2$. Les étapes 1 et 2 sont inutiles. On divise la ligne $p = 2$ par le pivot. On obtient : $P' = \begin{pmatrix} 1 & 0 & 3/5 \\ 0 & 1 & 1/5 \\ 0 & 0 & 1 \end{pmatrix}$ et $R' = \begin{pmatrix} 0{,}5 \\ 0 \\ 5/6 \end{pmatrix}$.

Combinaison linéaire $i \neq p$ :

- $i = 0$ : valeur $= P'_{i=0,p=2} = \frac{3}{5}$. D'où : ligne$(i=0) \leftarrow$ ligne $(i=0) -$ valeur $\times$ ligne$(p=2)$. On a alors : $P' = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1/5 \\ 0 & 0 & 1 \end{pmatrix}$ et $R' = \begin{pmatrix} \frac{1}{2} - \frac{3}{5} \times \frac{5}{6} \\ 0 \\ 5/6 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 5/6 \end{pmatrix}$.
- $i = 1$ : valeur $= P'_{i=1,p=2} = \frac{1}{5}$. D'où : ligne$(i=1) \leftarrow$ ligne $(i=1) -$ valeur $\times$ ligne$(p=2)$. On a alors : $P' = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ et $R' = \begin{pmatrix} 0 \\ -1/6 \\ 5/6 \end{pmatrix}$.

On a alors $Q = P'R' = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ -1/6 \\ 5/6 \end{pmatrix} = \begin{pmatrix} 0 \\ -1/6 \\ 5/6 \end{pmatrix}$.

On peut vérifier facilement que $Q = \begin{pmatrix} 0 \\ -1/6 \\ 5/6 \end{pmatrix}$ est solution du système de départ :

$$PQ = \begin{pmatrix} 2 & 4 & 2 \\ 1 & 4 & 2 \\ 2 & 9 & 3 \end{pmatrix} Q = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

# Interpolation polynomiale de Lagrange

L'interpolation consiste à construire une courbe à partir d'un nombre fini de points. La solution du problème d'interpolation doit passer nécessairement par les points initiaux. On utilisera par la suite les polynômes de Lagrange.

Soit $n$ un entier naturel non nul. On considere $(n + 1)$ points $\left(x_i, y_i\right)_{i \in [\![0,n]\!]}$ de $\mathbb{R}^2$. On suppose que les abscisses $x_i$ sont deux à deux distinctes. On définit les polynômes caractéristiques de Lagrange :

$$L_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}, i \in [\![0,n]\!]$$

On peut démontrer les propriétés suivantes :

- $L_i$ est un polynôme de degré $n$.
- $L_i(x_i) = 1$.
- $L_i(x_j) = 0,\ j \neq i$.

On considère $p$ le polynôme d'interpolation de Lagrange associé aux points $(x_0, y_0), (x_1, y_1), ..., (x_n, y_n)$ :

$$p(x) = \sum_{i=0}^{n} y_i L_i(x), x \in \mathbb{R}$$

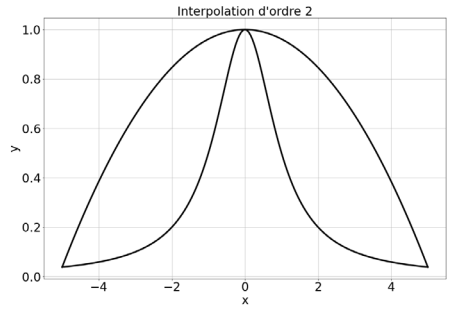
Soit la fonction $f$ définie par $f(x) = \frac{1}{1+x^2}, x \in \mathbb{R}$. On pose $a = -5$, $b = 5$, $h = \frac{b-a}{n}$ et $x_i = a + ih, i \in [\![0, n]\!]$.

Le programme suivant permet de représenter graphiquement la fonction $f$ et le polynôme d'interpolation d'ordre $n$ dans l'intervalle [–5, 5]. Les listes Python `xs` et `ys` contiennent respectivement les séries des abscisses $x_i$ et des ordonnées $y_i$.

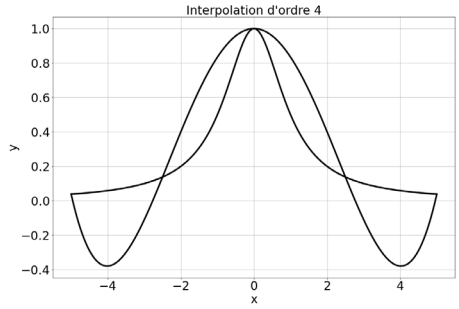
```
import matplotlib.pyplot as plt     # module matplotlib.pyplot renommé plt
def f(x):
    return 1/(1+x**2)
a=-5
b=5

# Interpolation d'ordre n. Le nombre de points vaut n+1
n=2
h=(b-a)/n
xs=[a+i*h for i in range(n+1)]      # liste des abscisses
ys=[f(xs[i]) for i in range(n+1)]   # liste des ordonnées

def lagrange(i, x, n):
    # i entier, x réel, n entier
    Li=1
    for j in range(n+1):            # j varie entre 0 inclus et n+1 exclu
        if j!=i:
            Li*=(x-xs[j])/(xs[i]-xs[j])
    return Li

def p(x, n):
    # x réel, n entier
    som=0
    for i in range(n+1):            # i varie entre 0 inclus et n+1 exclu
        som+=ys[i]*lagrange(i, x, n)
    return som

# Listes X et Y pour la représentation graphique de f
# N+1 nombre de points
N=200
H=(b-a)/N
X=[a+k*H for k in range(N+1)]       # liste des abscisses
```

```
Y=[f(X[k]) for k in range(N+1)]      # liste des ordonnées
YL=[p(X[k], n) for k in range(N+1)]

# Représentation graphique
plt.figure()                        # nouvelle fenêtre graphique
plt.plot(X, Y)                      # graphe Y en fonction de X
plt.plot(X, YL)                     # graphe YL en fonction de X
plt.show()                          # affiche le graphique
```

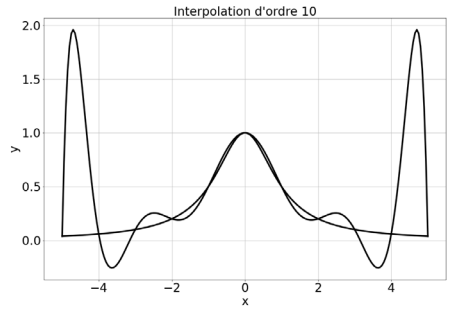
Interpolation d'ordre 2 :




Interpolation d'ordre 4 :




Interpolation d'ordre 10 :




Un phénomène d'oscillations apparaît et s'amplifie lorsque *n* augmente. Le phénomène de Runge peut être atténué en considérant une interpolation par morceaux (voir exercice 16.5 « Interpolation par morceaux »).

# Énoncés des exercices

## 16.1

**Équation différentielle du premier ordre (uniquement TSI et TPC)**

On considère l'équation différentielle $\frac{\mathrm{d}y}{\mathrm{d}t} = \frac{E - y}{\tau}$ avec $E$ et $\tau$ des constantes. La réponse $y(t)$ recherchée sur l'intervalle $[t_0, t_{\max}]$ sera obtenue par la méthode d'Euler. Le pas de calcul, noté $h$ sera choisi constant. L'intervalle de temps discrétisé est représenté par la liste $T = [t_0, t_1, ..., t_{N-1} = t_{\max}]$. Pour chaque instant $t_i$, une valeur approchée $y_i$ de la solution $y(t_i)$ de l'équation différentielle est recherchée. L'ensemble des $y_i$ est représenté par la liste `Y`.

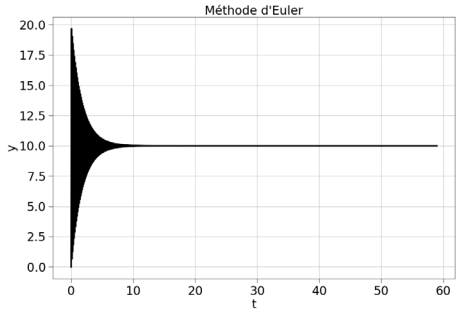
**a)** Écrire une relation de récurrence permettant de calculer $y_{i+1}$ en fonction de $y_i, E, h$ et $h$.

**b)** Écrire une fonction `Euler(y0, t0, h, N, E, tau)` qui admet comme arguments `y0`$= y(0)$, `t0` le temps initial de calcul, `h` le pas de calcul, `N` le nombre de points, `E` une constante et `tau` une constante. Cette fonction renverra les listes `T` et `Y`.

**c)** Écrire le programme principal permettant d'afficher graphiquement $y$ en fonction du temps $t$ avec les conditions suivantes : $t_0 = 0$, $y(0) = 0$, $E = 10$, $h = 0{,}2$ ms, $\tau = 30$ ms et $N = 1000$. Le graphique doit avoir les caractéristiques suivantes :

- affichage de « $t$ » pour l'axe des abscisses,
- affichage de « $y$ » pour l'axe des ordonnées,
- affichage du titre « Méthode d'Euler ».

**d)** Commenter la courbe obtenue avec $h = 59$ ms, les autres conditions précédentes demeurant inchangées.




## 16.2

**Équation différentielle du deuxième ordre (uniquement TSI et TPC)**

On cherche à résoudre numériquement l'équation différentielle du deuxième ordre : $\frac{\mathrm{d}^2u}{\mathrm{d}t^2} + b\frac{\mathrm{d}u}{\mathrm{d}t} + cu = 0$ (1).

On pose $g(t) = \frac{\mathrm{d}u(t)}{\mathrm{d}t}$ et $Y$ tel que $Y(t) = \begin{pmatrix} u(t) \\ g(t) \end{pmatrix}$. On cherche à mettre l'équation différentielle (1) sous la forme du problème de Cauchy : $\frac{\mathrm{d}Y}{\mathrm{d}t} = F(Y(t), t)$. On pose $Y(0) = Y_0 = \begin{pmatrix} u(0) \\ g(0) \end{pmatrix}$.

La réponse $u(t)$ recherchée sur l'intervalle $[0, t_{max}]$ sera obtenue par la méthode d'Euler explicite. Le pas de calcul, noté $h$, sera choisi constant. L'intervalle de temps discrétisé est alors représenté par la liste $T = [t_0 = 0, t_1, ... t_{N-1} = t_{max}]$. Pour chaque instant $t_i$, une valeur approchée $Y_i$ de la solution $Y(t_i)$ de l'équation différentielle est recherchée. On utilisera 3 listes `T`, `U` et `G` comportant chacune $N$ éléments.

**a)** Déterminer la fonction $F(Y(t), t)$.

**b)** Donner la relation de récurrence qui lie $Y_{i+1}$ à $Y_i, F(t_i, Y_i)$ et au pas de calcul $h$.

**c)** Écrire une fonction `F(Yi, ti)` qui admet comme arguments `Yi` la valeur du vecteur `Y` au temps discrétisé $t_i$, `ti` la valeur du temps discrétisé, et qui retourne la valeur de $F(Y_i(t_i), t_i)$.

**d)** Écrire une fonction `Euler(Yini, h, tmax, F)` qui prend comme arguments `Yini` une liste contenant $Y_0$ la condition initiale, `h` le pas de calcul, `tmax` le temps final de calcul et `F` la fonction du problème de Cauchy. Cette fonction retourne les listes `T`, `U` et `G`.

Écrire le programme principal permettant de résoudre l'équation différentielle (1) avec $b$ = 0,75, $c$ = 36, $t_{max}$ = 10 s, $h$ = 0,6 ms, $u(0) = 2$ et $g(0) = 0$. Représenter graphiquement $u$ en fonction de $t$.

## 16.3

**Résolution d'un système linéaire par la méthode de Gauss (uniquement TSI et TPC)**

On étudie un moteur thermique constitué de quatre pistons. Depuis l'intégration des calculateurs dans l'automobile, les principaux paramètres de commande du moteur à allumage commandé tels que l'ouverture du papillon, la durée d'injection, l'avance à l'allumage, etc. sont contrôlés numériquement par des cartographies et/ou des boucles d'asservissement.

La quantité de carburant injectée est directement corrélée à la durée d'ouverture des injecteurs, que l'on note **durée d'injection**. Dans les conditions de fonctionnement stabilisé, le dosage de base est le résultat d'une interpolation cartographique calculée à partir de la vitesse et de la charge du moteur. Toutes les cartographies et tous les programmes du moteur sont stockés sous forme de fichiers dans la mémoire morte du calculateur (ROM), qui dispose de 32 ko d'espace. Au démarrage du véhicule, le programme de gestion du moteur et certaines données seront chargés dans la mémoire vive (RAM), qui dispose de 3 ko d'espace.

On s'intéresse au traitement de la cartographie qui permet de déterminer la durée d'injection. Le tableau 1 représente l'affichage d'une cartographie des durées d'injection pour un moteur 4 temps.

| | | Pression au collecteur P en bar | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | **0,295** | **0,39** | **0,48** | **0,565** | **0,645** | **0,72** | **0,79** |
| **Vitesse de rotation ω en tr/min** | **900** | 2,702 | 3,776 | 4,852 | 5,9 | 6,962 | 8,004 | 9,036 |
| | **1300** | 3,064 | 4,162 | 5,248 | 6,33 | 7,734 | 8,774 | 9,766 |
| | **1700** | 3,27 | 4,412 | 5,552 | 6,644 | 7,734 | 8,774 | 9,766 |
| | **2200** | 3,462 | 4,63 | 5,804 | 6,952 | 8,062 | 9,126 | 10,154 |
| | **2700** | 3,498 | 4,71 | 5,916 | 7,09 | 8,23 | 9,334 | 10,39 |

**Tableau 1 : Cartographie des durées d'injection en ms.**

La cartographie est alors chargée en mémoire sous la forme suivante :

- La première ligne (0,295 ; 0,39 ; 0,48…), qui contient des valeurs de pression à l'admission en bar, est stockée dans une liste de valeurs : $P = \left[ P_j \right]_{0 \le j \le M-1}$.
- La première colonne (900 ; 1300 ; 1700…), qui contient des valeurs de la vitesse de rotation moteur en tr/min, est stockée dans une liste de valeurs : $OMEGA = \left[ OMEGA_i \right]_{0 \le i \le N-1}$.
- Les durées d'injection sont stockées sous forme d'une liste de listes `T`. On peut lire pour chaque couple de valeurs de pression et de rotation moteur la valeur de la durée d'injection correspondante `T[i][j]` en ms.

Le capteur de pression au collecteur d'admission mesure une valeur de pression notée `Pcol`. La vitesse de rotation du moteur mesurée est notée `OMEGAmot`. Pour un couple de valeurs pression-vitesse de rotation (`Pcol`, `OMEGAmot`) quelconque, le calculateur doit alors déterminer les valeurs de la durée d'injection en réalisant une interpolation à partir des valeurs de la liste de listes `T`.

**Étape 1 :**

Le calculateur doit déterminer les indices `i` et `j` tels que $P\left[ j \right] \le Pcol \le P\left[ j+1 \right]$ et $OMEGA\left[ i \right] \le OMEGAmot \le OMEGA\left[ i+1 \right]$ .

**a)** Écrire une fonction `indice(A, val)` qui prend pour arguments une liste notée `A` et un réel noté `val`. La liste `A` est triée par ordre croissant. La fonction doit retourner un entier `id` tel que : $A\left[ id \right] \le val \le A\left[ id+1 \right]$. On supposera que `id` existe toujours.

Le calculateur doit ensuite lire dans la liste de listes `T` les durées d'injection correspondant aux indices `i` et `j` précédemment déterminés.

**b)** Écrire une fonction `extraire(T, P, OMEGA, i, j)` qui prend pour arguments la liste de listes `T`, les listes `P` et `OMEGA`, et deux entiers `i` et `j`. La fonction doit retourner une liste de listes `ST` :

$$ST = \left[ \left[ T[i,j], T[i,j+1], \left[ P[j], \text{OMEGA}[i] \right], \right. \right.$$
$$\left. \left. \left[ T[i+1,j], T[i+1,j+1], P[j+1], \text{OMEGA}[i+1] \right] \right] \right.$$

**c)** Écrire la suite d'instructions qui, à partir des variables `OMEGAmot`, `Pcol`, des listes *P* et `OMEGA` et de la liste de listes `T`, permet de déterminer la liste de listes `ST` telle que définie à la question précédente.

**Étape 2 :**

Une fois les quatre valeurs de durée d'injection déterminées, il est nécessaire de calculer une durée d'injection *t* à partir d'une interpolation bilinéaire. L'interpolation bilinéaire de la fonction de deux variables $t(x,y)$ s'écrit comme suit :

$$t(x,y) = b_0 + b_1 x + b_2 y + b_3 xy \ ; \ x \in \left[ 0,1 \right] \text{ et } y \in \left[ 0,1 \right]$$

où $b_0$, $b_1$, $b_2$ et $b_3$ sont les inconnues du problème.

La détermination des coefficients $b_{i,i\in[0,3]}$ est un problème linéaire qui doit être résolu à chaque pas de temps et pour chaque cartographie sous la forme :

$$PQ = R \text{ et } Q = P^{-1}Q$$

*P* est une matrice carrée de dimension $n \times n$ ; *Q* et *R* sont des matrices colonnes de dimension *n*.

On envisage dans un premier temps l'utilisation de la méthode de Gauss avec recherche du pivot partiel pour résoudre le système linéaire et obtenir la matrice *Q*.

**d)** Écrire l'algorithme du pivot de Gauss permettant d'obtenir la matrice *Q*. On admettra que les termes diagonaux sont tous non nuls.

**e)** Quelle est la complexité du pivot de Gauss en fonction de la dimension de la matrice ? Justifier votre réponse.

On posera par la suite : $x = \dfrac{Pcol - P[j]}{P[j+1] - P[j]}$ et $y = \dfrac{OMEGAmot - OMEGA[i]}{OMEGA[i+1] - OMEGA[i]}$.

On connaît les valeurs de $t(x,y)$ pour quatre couples de valeurs $(x,y)$ par lecture de la cartographie :

$$\begin{pmatrix} t(x=0,y=0) = ST[0,0] & t(x=1,y=0) = ST[0,1] \\ t(x=0,y=1) = ST[1,0] & t(x=1,y=1) = ST[1,1] \end{pmatrix}$$

**f)** Montrer que le problème à résoudre devient :

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} ST[0,0] \\ ST[0,1] \\ ST[1,0] \\ ST[1,1] \end{pmatrix}$$

**g)** Montrer que la résolution par substitution s'écrit :

$$\begin{aligned} b_0 &= ST[0,0] \\ b_1 &= ST[0,1] - ST[0,0] \\ b_2 &= ST[1,0] - ST[0,0] \\ b_3 &= ST[1,1] - ST[0,1] - ST[1,0] + ST[0,0] \end{aligned}$$

**h)** Comparer la complexité de cet algorithme à celui présenté à la question d).

**i)** Écrire une fonction `interpol(ST, Pcol, OMEGAmot)` qui retourne une valeur de la durée d'injection obtenue par une interpolation bilinéaire des éléments de la table.

## 16.4

■■□□

**Interpolation polynomiale de Lagrange (uniquement TSI et TPC)**

On considère une fonction $f$ définie par $f(x) = x\sin(\pi x), x \in \mathbb{R}$. On cherche à représenter graphiquement la fonction $f$ et le polynôme d'interpolation de degré $n$ par rapport aux $n + 1$ points équidistants dans l'intervalle $[-1,1]$. On note $f_i = f(x_i), i \in [\![0,n]\!]$. Les abscisses $x_i$ et les ordonnées $f_i$ sont stockées respectivement dans les listes `xs` et `ys`.

**a)** Écrire une fonction `lagrange` qui admet comme arguments un entier `i`, un réel `x` et un entier `n`. La fonction retourne $L_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \dfrac{x - x_j}{x_i - x_j}, i \in [\![0,n]\!]$.

**b)** Écrire une fonction `poly` qui admet comme arguments un réel `x` et un entier `n`. La fonction retourne $p(x) = \sum_{i=0}^{n} f_i L_i(x), \ x \in \mathbb{R}$.

**c)** Écrire le programme principal permettant de représenter graphiquement la fonction $f$ et le polynôme d'interpolation de Lagrange d'ordre $n = 6$ dans l'intervalle [–1,5 ; 1,5].

**16.5**

**Interpolation par morceaux (uniquement TSI et TPC)**

On considère une fonction $f$ définie par $f(x)=\dfrac{1}{1+x^2}, x\in\mathbb{R}$. On cherche à représenter graphiquement la fonction $f$, le polynôme d'interpolation de degré $n$ ainsi que l'interpolation linéaire par morceaux par rapport aux $n+1$ points équidistants dans l'intervalle $[a,b]=[-5,5]$. On définit une partition de $[a,b]$ en $n$ sous-intervalles $I_i=[x_i,x_{i+1}], i\in[\![0,n-1]\!]$, de longueur $h=\dfrac{b-a}{n}$ avec $x_i=a+ih, i\in[\![0,n]\!]$. On note $f_i=f(x_i), i\in[\![0,n]\!]$. Pour $i\in[\![0,n-1]\!]$, on définit $p_{1,i}$ le polynôme d'interpolation linéaire de Lagrange aux nœuds $(x_i,f_i),(x_{i+1},f_{i+1})$. Le polynôme d'interpolation par morceaux $p_1^h$ est défini, pour tout $i\in[\![0,n-1]\!]$, par :

$$p_i^h(x)=p_{1,i}(x), \forall x\in I_i$$

Le polynôme $p_{1,i}$ est un polynôme de degré 1, pour tout $i\in[\![0,n-1]\!]$.

Les abscisses $x_i$ et les ordonnées $f_i$ sont stockées respectivement dans les listes `xs` et `ys`.

**a)** Donner l'expression de $p_{1,i}$ sur chaque intervalle $I_i$, pour $i\in[\![0,n-1]\!]$.

**b)** Écrire une fonction `interpol` qui admet comme argument `x` et retourne $p_1^h(x), x\in[a,b]$.

Écrire le programme principal permettant de représenter graphiquement la fonction $f$, le polynôme d'interpolation de degré $n=6$ ($p(x)=\sum_{i=0}^{n} f_i L_i(x)$, $x\in\mathbb{R}$ avec $L_i(x)=\prod_{\substack{j=0\\ j\neq i}}^{n}\dfrac{x-x_j}{x_i-x_j}$, $i\in[\![0,n]\!]$) et une interpolation linéaire par morceaux pour $n=6$ dans l'intervalle $[a,b]$.

# Du mal à démarrer ?

**16.1**

**a)** Utiliser la méthode d'Euler explicite.

**b)** Utiliser une boucle `for` et `Y.append` pour ajouter à chaque étape un élément à la liste `Y`.

**c)** Utiliser `plt.plot`, `plt.title`, `plt.xlabel`, `plt.ylabel`.

**16.2**

**a)** Transformer l'équation différentielle en un système de deux équations différentielles du premier ordre.

**b)** Utiliser la méthode d'Euler pour évaluer $\left(\dfrac{\mathrm{d}Y}{\mathrm{d}t}\right)_{t_i}$.

**c)** La fonction `F` retourne une liste de deux éléments.

**d)** Utiliser une boucle `for` avec $N-1$ étapes.

**16.3**

**a)** Utiliser une boucle `while`.

**b)** Utiliser `T[i][j]` pour les listes de listes.

**d)** Utiliser `copy.deepcopy(P)` pour ne pas modifier `P`.

**e)** Calculer le nombre d'opérations élémentaires.

**16.4**

**a)** Initialiser une variable `Li` à 1. Utiliser une boucle `for` en faisant varier `j` entre 0 inclus et `n+1` exclu.

**b)** Initialiser une variable `som` à 1. Utiliser une boucle `for` en faisant varier `i` entre 0 inclus et `n+1` exclu.

**c)** Poser `N=200` et utiliser `N+1` points pour la représentation graphique.

**16.5**

**a)** Déterminer l'équation de la droite passant par les points $(x_i, f_i), (x_{i+1}, f_{i+1})$.

**b)** Utiliser les fonctions `lagrange` et `poly` définies dans l'exercice 16.4 « Interpolation polynomiale de Lagrange ». Poser `N=200` et utiliser `N+1` points pour la représentation graphique.

# Corrigés des exercices

**16.1**

**a)** La méthode d'Euler explicite permet d'écrire : $\left(\frac{\mathrm{d}y}{\mathrm{d}t}\right)_{t_i} \approx \frac{y_{i+1}-y_i}{h} = \frac{E-y_i}{\tau}$. On en déduit la relation de récurrence : $y_{i+1} = y_i + h\frac{E-y_i}{\tau}$.

**Remarque**

Sans indication contraire de l'énoncé, on utilise la méthode d'Euler explicite.

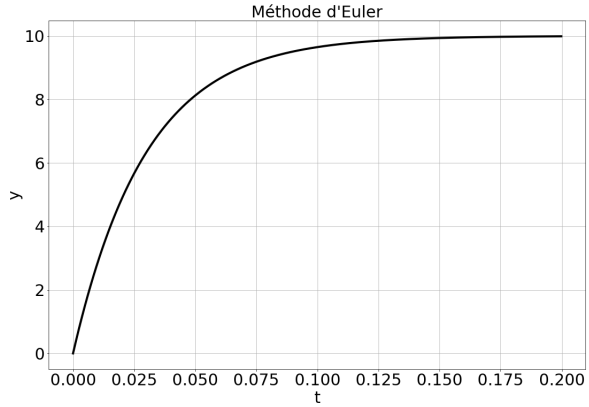
**b)**

```
def Euler(y0, t0, h, N, E, tau):
    T=[t0]                              # 1er élément de la liste T
    Y=[y0]                              # 1er élément de la liste Y
    for i in range(N-1):                # i varie entre 0 inclus et N-1 exclu
        T.append(T[i]+h)                # ajout d'un élément à la liste T
        Y.append(Y[i]+h*(E-Y[i])/tau)   # ajout d'un élément à la liste Y
    return T, Y
```

**c)**

```
import matplotlib.pyplot as plt # bibliothèque pyplot renommée en plt

N=1000                          # nombre de points
h=0.2e-3                        # intervalle entre deux instants consécutifs
t0=0
y0=0
E=10
tau=30e-3
T, Y=Euler(y0, t0, h, N, E, tau)

plt.figure()                    # nouvelle fenêtre graphique
plt.plot(T, Y)                  # Y en fonction de T
plt.title("Méthode d'Euler")    # titre
```

```
plt.xlabel("t")                # axe des abscisses
plt.ylabel("y")                # axe des ordonnées
plt.show()                     # affiche le graphique
```




**d)** La solution obtenue avec la méthode d'Euler ne correspond pas du tout à la solution de l'équation différentielle étudiée. La qualité de la solution obtenue dépend du pas.

**Remarque**

Si on considère l'équation différentielle $\frac{dy}{dt} = \lambda y$, On peut écrire que $y_{i+1} = y_i + h\lambda y_i = y_i\left(1+h\lambda\right)$. Pour que la méthode soit stable, il faut que $\left|1+h\lambda\right| < 1$. On peut en déduire un critère de stabilité sur le pas.

## 16.2

**a)** On pose $g(t) = \frac{du}{dt}$. L'équation différentielle (1) s'écrit alors sous la forme : $\frac{dg}{dt} + bg + cu = 0$, soit $\frac{dg}{dt} = -cu - bg$. On transforme alors l'équation différentielle (1) en un système de deux équations différentielles du premier ordre :

$$\begin{pmatrix} \frac{du}{dt} \\ \frac{dg}{dt} \end{pmatrix} = \begin{pmatrix} g(t) \\ -cu(t) - bg(t) \end{pmatrix}$$

On a donc $\frac{dY}{dt} = F\left(Y(t),t\right)$ avec $Y(t) = \begin{pmatrix} u(t) \\ g(t) \end{pmatrix}$ et $F\left(Y(t),t\right) = \begin{pmatrix} g(t) \\ -cu(t) - bg(t) \end{pmatrix}$.

À l'instant $t_i$, on a : $F\left(Y_i(t_i),t_i\right) = \begin{pmatrix} g_i \\ -cu_i - bg_i \end{pmatrix}$. Il faut donc retourner la liste : $\left[g_i, -cu_i - bg_i\right]$.

**b)** Pour évaluer numériquement $\left(\frac{dY}{dt}\right)_{t_i}$ à l'instant $t_i$, on utilise la méthode d'Euler explicite :

$$\left(\frac{dY}{dt}\right)_{t_i} \approx \frac{Y\left(t_{i+1}\right) - Y\left(t_i\right)}{h}$$

On a alors : $\frac{Y(t_{i+1})-Y(t_i)}{h}=F\left(Y(t_i),t_i\right)$. La relation de récurrence s'écrit donc :

$Y_{i+1}=Y_i+h\left(F\left(Y_i,t_i\right)\right)$, soit $\begin{cases}u_{i+1}=u_i+hg_i\\ g_{i+1}=g_i+h\left(-cu_i-bg_i\right)\end{cases}$

**Remarques**

- La relation de récurrence s'écrit : $Y_{i+1}=Y_i+h\left(F\left(Y_i,t_i\right)\right)$. On a un schéma explicite puisqu'on calcule le point suivant en fonction du point précédent.
- Dans un schéma implicite, on a $Y_{i+1}=Y_i+h\left(F\left(Y_{i+1},t_{i+1}\right)\right)$, soit $\frac{Y_{i+1}-Y}{h}=F\left(Y_{i+1},t_{i+1}\right)$. Dans ce cas, la nouvelle valeur $Y_{i+1}$ est calculée en utilisant la valeur de la dérivée à la nouvelle position.

**c)** On a vu que $F\left(Y(t),t\right)=\begin{pmatrix}g(t)\\ -cu(t)-bg(t)\end{pmatrix}$ et $Y(t)=\begin{pmatrix}u(t)\\ g(t)\end{pmatrix}$.

À l'instant $t_i$, on a : $F\left(Y_i(t_i),t_i\right)=\begin{pmatrix}g_i\\ -cu_i-bg_i\end{pmatrix}$. Il faut donc retourner la liste : $\left[g_i,-cu_i-bg_i\right]$.

- On récupère $u_i$ avec le premier élément de la liste `Yi` qui est en argument d'entrée de la fonction, soit $u_i=Y_i[0]$.
- On récupère $g_i$ avec le deuxième élément de la liste `Yi` qui est en argument d'entrée de la fonction, soit $g_i=Y_i[1]$.

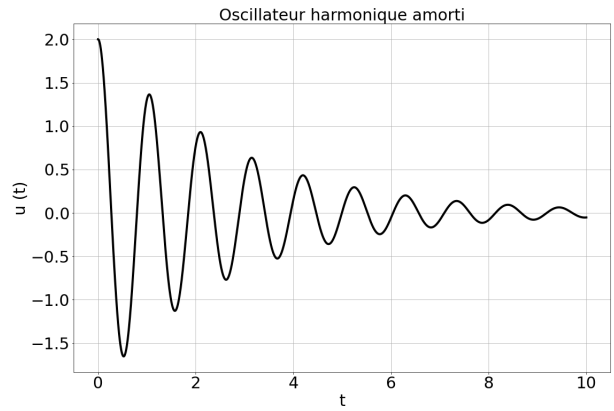
```
import matplotlib.pyplot as plt   # bibliothèque pyplot renommée en plt

def F(Yi, ti):
    # Yi[0] représente U[i] et Yi[1] représente G[i]
    return Yi[1], -c*Yi[0]-b*Yi[1]
```

**d)** Il faut bien définir le nombre de points. Comme $t_{\max}=(N-1)h$, alors $N=\text{int}\left(1+\frac{t_{\max}}{h}\right)$. Il faut prendre la valeur entière sinon il y a un problème de type pour la syntaxe `for i in range(N-1)`.

```
def Euler(Yini, h, tmax, F):
    N=int(1+tmax/h)               # nombre de points de discrétisation
    T=[0]                         # 1er élément de la liste T
    U=[Yini[0]]                   # 1er élément de la liste U
    G=[Yini[1]]                   # 1er élément de la liste G
    for i in range(N-1):
        Y=[U[i], G[i]]
        resF=F(Y, T[i])           # résultat de l'appel de F
        U.append(U[i]+h*resF[0])  # ajout d'un élément à la liste U
        G.append(G[i]+h*resF[1])  # ajout d'un élément à la liste G
        T.append(T[i]+h)          # on incrémente de h à chaque boucle
    return T, U, G

# Initialisation des variables
Yini=[2, 0]
```

```
b=0.75
c=36
tmax=10
h=0.6e-3
N=int(1+tmax/h)                # on a : tmax=(N-1)h
T, U, H=Euler(Yini, h, tmax, F)

plt.figure()
plt.plot(T, U)                 # représentation graphique de U en fonction de T
plt.xlabel('t')
plt.ylabel('u(t)')
plt.title("Oscillateur harmonique amorti")
plt.show()
```

On obtient le graphe :




## 16.3

**a)**

```
def indice(A, val):
    id=0
    while A[id] > val or val > =A[id+1]:
        id+=1
    return id
    # suppose que id existe toujours sinon
    # la boucle pourrait ne pas se terminer
```

Autre possibilité pour la fonction indice :

```
def indice(A, val):
    id=0
    n=len(A)
    for id in range(n-1):          # id varie entre 0 inclus et n-1 exclu
        if A[id]<=val and val<A[id+1]:
            return id              # on quitte la boucle for
```

**b)**

```
def extraire(T, P, OMEGA, i, j):
    ST=[]       # création d'une liste vide
    ST.append([T[i][j],T[i][j+1],P[j],OMEGA[i]])
    ST.append([T[i+1][j],T[i+1][j+1],P[j+1],OMEGA[i+1]])
    return ST
```

**c)**

```
j=indice(P, Pcol)
i=indice(OMEGA, OMEGAmot)
ST2=extraire2(T2, P, OMEGA, i, j)   # liste de listes
```

On obtient ST = [[6.644, 7.734, 0.565, 1700], [6.952, 8.062, 0.645, 2200]].

**d)**

```
import copy

def rec_pivot(A, p):
    n=len(A)
    i, pivot=p, A[p][p]
    for k in range(p, n):           # k varie entre p inclus et n exclu
        if A[k][p]>pivot:
            i, pivot=k, A[k][p]
    return i, pivot

def permut_ligne(A, i, p):
    n=len(A)
    for k in range(n):
        val=A[i][k]
        A[i][k]=A[p][k]
        A[p][k]=val

def gauss(P, R):
    Pc=copy.deepcopy(P)             # on ne modifie pas P
    Rc=copy.deepcopy(R)             # on ne modifie pas R
    n=len(Pc)
    for p in range(0, n):           # p varie entre 0 inclus et n-1 exclu
        i, pivot=rec_pivot(Pc, p)   # recherche du pivot
        if i>p:                     # permutation des lignes i et p si i > p
            permut_ligne(Pc, i, p)
            Rc[i], Rc[p]=Rc[p], Rc[i]
        # division de la ligne p par le pivot
        for j in range(n):          # j varie entre 0 inclus et n exclu
            Pc[p][j]=Pc[p][j]/pivot
        Rc[p]=Rc[p]/pivot
        # combinaison linéaire
        for i in range(n):
```

```
            if i!=p:
                coeff=Pc[i][p]
                for j in range(n):  # j varie entre 0 inclus et n exclu
                    Pc[i][j]=Pc[i][j]-coeff*Pc[p][j]
                Rc[i]=Rc[i]-coeff*Rc[p]
    # solution Q
    Q=[Pc[i][i]*Rc[i] for i in range(n)]
    return Q

P=[[2, 4, 2], [1, 4, 2], [2, 9, 3]]
R=[1, 1, 1]
Q=gauss(P, R)
```

**e)** On se place dans le pire des cas pour calculer la complexité.

On fait varier $p$ entre 0 et $n-1$. Pour chaque valeur de $p$ :

- Recherche du pivot : $4+\left(n-1-p\right)\times 4=4n-4p$ opérations élémentaires.
- Permutation des lignes $i$ et $p$ : $3+3n$ opérations élémentaires.
- Division de la ligne $p$ par le pivot : $3n+3$ opérations élémentaires.
- Combinaison linéaire : $n\times\left(2+4n+4\right)=6n+4n^2$ opérations élémentaires.

Solution $Q$ : $4n$ opérations élémentaires.

On a donc $\sum_{p=0}^{n-1}\left(4n-4p+3+3n+3n+3+6n+4n^2\right)+4n=4n^3+14n^2+8n$ opérations élémentaires.

La complexité est en $O\left(n^3\right)$.

**f)** On pose : $t\left(x,y\right)=b_0+b_1x+b_2y+b_3xy$ ; $x\in\left[0,1\right]$ et $y\in\left[0,1\right]$.

On en déduit que : $t\left(0,0\right)=b_0$ ; $t\left(1,0\right)=b_0+b_1$ ; $t\left(0,1\right)=b_0+b_2y$ et $t\left(x,y\right)=b_0+b_1+b_2+b_3$.

On a bien $\begin{pmatrix}1&0&0&0\\1&1&0&0\\1&0&1&0\\1&1&1&1\end{pmatrix}\begin{pmatrix}b_0\\b_1\\b_2\\b_3\end{pmatrix}=\begin{pmatrix}b_0\\b_0+b_1\\b_0+b_2\\b_0+b_1+b_2+b_3\end{pmatrix}=\begin{pmatrix}t\left(0,0\right)\\t\left(1,0\right)\\t\left(0,1\right)\\t\left(1,1\right)\end{pmatrix}=\begin{pmatrix}ST\left[0,0\right]\\ST\left[0,1\right]\\ST\left[1,0\right]\\ST\left[1,1\right]\end{pmatrix}$.

**g)** On résout directement ce système :

- $b_0=ST\left[0,0\right]$ ;
- $b_1=ST\left[0,1\right]-b_0=ST\left[0,1\right]-ST\left[0,0\right]$ ;
- $b_2=ST\left[1,0\right]-b_0=ST\left[1,0\right]-ST\left[0,0\right]$ ;
- $b_3=ST\left[1,1\right]-b_0-b_1-b_2=ST\left[1,1\right]-ST\left[0,0\right]-\left(ST\left[0,1\right]-ST\left[0,0\right]\right)-\left(ST\left[1,0\right]-ST\left[0,0\right]\right)$.

Après simplification, on a : $b_3=ST\left[1,1\right]+ST\left[0,0\right]-ST\left[0,1\right]-ST\left[1,0\right]$.

On obtient finalement :

$b_0=ST\left[0,0\right]$

$b_1=ST\left[0,1\right]-ST\left[0,0\right]$

$b_2=ST\left[1,0\right]-ST\left[0,0\right]$

$b_3=ST\left[1,1\right]-ST\left[0,1\right]-ST\left[1,0\right]+ST\left[0,0\right]$

**h)** Pour cette matrice triangulaire inférieure, on peut très facilement calculer les coefficients $b_i$.

La première ligne permet de calculer $b_0$. La deuxième ligne permet d'en déduire $b_1$ et ainsi de suite pour tous les coefficients $b_i$.

On se place dans le pire des cas. On fait varier $i$ entre 0 entre $n-1$. Pour chaque étape $i$, on a au maximum $i+1$ opérations élémentaires.

Le nombre d'opérations élémentaires vaut donc : $\sum_{i=0}^{n-1}(i+1) = \frac{n(n-1)}{2} + n$.

La complexité est quadratique en $O(n^2)$. Elle est plus faible qu'à la question d) puisqu'on a une matrice triangulaire.

**i)**

```
def interpol(ST, Pcol, OMEGAmot):
    P=[[1,0,0,0], [1,1,0,0], [1,0,1,0], [1,1,1,1]]
    R=[ST[0][0], ST[0][1], ST[1][0], ST[1][1]]
    Q=gauss(P, R)
    x=(Pcol-ST[0][2])/(ST[1][2]-ST[0][2])
    y=(OMEGAmot-ST[0][3])/(ST[1][3]-ST[0][3])
    t=Q[0]+Q[1]*x+Q[2]*y+Q[3]*x*y
    return t
```

**Remarque**

Voici un exemple de programme principal :

```
P=[0.295, 0.39, 0.48, 0.565, 0.645, 0.72, 0.79]
OMEGA=[900, 1300, 1700, 2200, 2700]
T=[[2.702, 3.776, 4.852, 5.9, 6.962, 8.004, 9.036],
[3.064, 4.162, 5.248, 6.33, 7.734, 8.774, 9.766],
[3.27, 4.412, 5.552, 6.644, 7.734, 8.774, 9.766],
[3.462, 4.63, 5.804, 6.952, 8.062, 9.126, 10.154],
[3.498, 4.71, 5.916, 7.09, 8.23, 9.334, 10.39]]
Pcol=0.60
OMEGAmot=1750
j=indice(P, Pcol)
i=indice(OMEGA, OMEGAmot)
ST=extraire(T, P, OMEGA, i, j)
t=interpol(ST, Pcol, OMEGAmot)
print("durée d'injection = ", t)
```

## 16.4

**a)**

```
def lagrange(i, x, n):
    # i entier, x réel, n entier
    Li=1
```

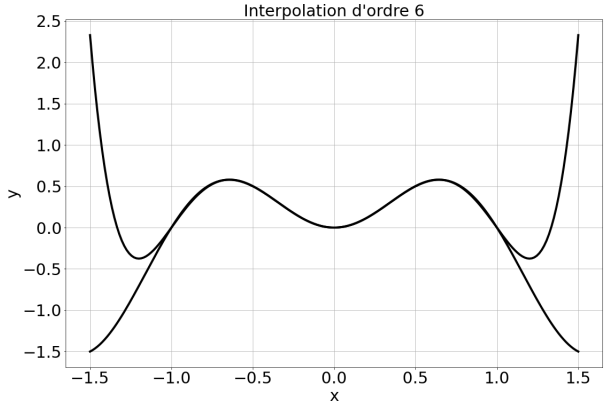
```
    for j in range(n+1):          # j varie entre 0 inclus et n+1 exclu
        if j!=i:
            Li*=(x-xs[j])/(xs[i]-xs[j])
    return Li
```

**b)**

```
def poly(x, n):
    # x réel, n entier
    som=0
    for i in range(n+1):          # i varie entre 0 inclus et n+1 exclu
        som+=ys[i]*lagrange(i, x, n)
    return som
```

**c)**

```
import math as m                  # module math renommé m
import matplotlib.pyplot as plt   # module matplotlib.pyplot renommé plt

def f(x):
    return x*m.sin(m.pi*x)
a=-1
b=1

# Interpolation d'ordre n. Le nombre de points vaut n+1
n=6
h=(b-a)/n
xs=[a+i*h for i in range(n+1)]    # liste des abscisses
ys=[f(xs[i]) for i in range(n+1)] # liste des ordonnées

# Listes X et Y pour la représentation graphique de f
N=200                             # N+1 nombre de points
A=-1.5
B=1.5
H=(B-A)/N
X=[A+k*H for k in range(N+1)]     # liste des abscisses
Y=[f(X[k]) for k in range(N+1)]   # liste des ordonnées
YL=[poly(X[k], n) for k in range(N+1)]

# Représentation graphique
plt.figure()                      # nouvelle fenêtre graphique
plt.plot(X, Y)                    # graphe Y en fonction de X
plt.plot(X, YL)                   # graphe YL en fonction de X
plt.show()                        # affiche le graphique
```




## 16.5

**a)** On considère une loi affine entre deux points successifs. Soit $i \in [\![0, n-1]\!]$, $\forall x \in [x_i, x_{i+1}]$ : $y = y_i + (x - x_i)\dfrac{y_{i+1} - y_i}{x_{i+1} - x_i}$. Pour l'intervalle $I_i$, on en déduit la fonction polynôme définie par $p_{1,i}(x) = y_i + (x - x_i)\dfrac{y_{i+1} - y_i}{x_{i+1} - x_i}$.

**b)** On pose $h = \dfrac{b-a}{n}$ et $x_i = a + ih, i \in [\![0, n]\!]$.

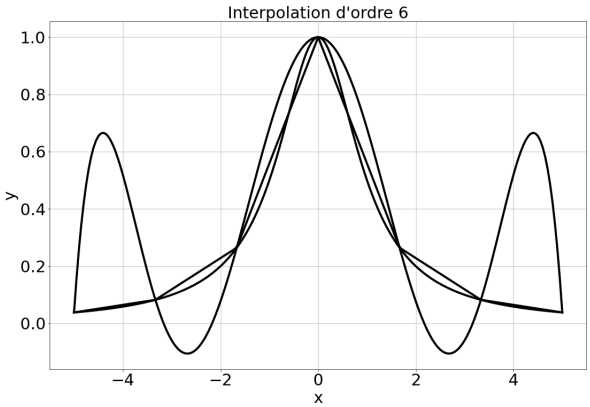
Soit $x \in [a, b]$. On cherche un entier *i* tel que *x* appartienne au sous-intervalle $I_i = [x_i, x_{i+1}], i \in [\![0, n-1]\!]$.

Avec Python, on obtient *i* en calculant $\text{int}\left(\dfrac{x - x_0}{h}\right)$.

- Si $x = x_0 + \dfrac{3h}{2}$, alors $\text{int}\left(\dfrac{x - x_0}{h}\right) = \text{int}\left(\dfrac{3}{2}\right) = 1$, *x* est bien dans l'intervalle $I_1 = [x_1, x_2]$.
- Si $x = x_n$, alors $\text{int}\left(\dfrac{x_n - x_0}{h}\right) = n$. Dans ce cas, la fonction retourne directement $y_n$ car l'intervalle $[x_n, x_{n+1}]$ n'est pas défini.

```
import math as m                    # module math renommé m
import matplotlib.pyplot as plt     # module matplotlib.pyplot renommé plt

def f(x):
    return 1/(1+x**2)
a=-5
b=5

# Interpolation d'ordre n. Le nombre de points vaut n+1
n=6
h=(b-a)/n
xs=[a+i*h for i in range(n+1)]      # liste des abscisses
ys=[f(xs[i]) for i in range(n+1)]   # liste des ordonnées
```

```
def lagrange(i, x, n):
    # i entier, x réel, n entier
    Li=1
    for j in range(n+1):            # j varie entre 0 inclus et n+1 exclu
        if j!=i:
            Li*=(x-xs[j])/(xs[i]-xs[j])
    return Li

def poly(x, n):
    # x réel, n entier
    som=0
    for i in range(n+1):            # i varie entre 0 inclus et n+1 exclu
        som+=ys[i]*lagrange(i, x, n)
    return som

def interpol(x):                    # x compris entre a et b
    i=int((x-xs[0])/h)
    if x!=xs[n]:
        # x est dans l'intervalle Ii=[xi, xi+1]
        return ys[i]+(x-xs[i])*(ys[i+1]-ys[i])/(xs[i+1]-xs[i])
    else:
        # l'intervalle [xn, xn+1] n'est pas défini
        return ys[n]

# Listes X et Y pour la représentation graphique de f
# polynôme d'interpolation de Lagrange
N=200                               # N+1 nombre de points
H=(b-a)/N
X=[a+k*H for k in range(N+1)]       # liste des abscisses
Y=[f(X[k]) for k in range(N+1)]     # liste des ordonnées
YL=[poly(X[k], n) for k in range(N+1)]

# Liste Ym pour la représentation graphique de f
# interpolation linéaire par morceaux
Y=[f(X[k]) for k in range(N+1)]     # liste des ordonnées
Ym=[interpol(X[k]) for k in range(N+1)]

# Représentation graphique
plt.figure()                        # nouvelle fenêtre graphique
plt.plot(X, Y)                      # graphe Y en fonction de X
plt.plot(X, YL)                     # graphe YL en fonction de X
plt.plot(X, Ym)                     # graphe Ym en fonction de X
plt.show()                          # affiche le graphique
```




On observe des oscillations avec l'interpolation polynomiale de Lagrange de degré $n = 6$ : c'est le phénomène de Runge qui peut être atténué en considérant une interpolation par morceaux.

# Index