NOUVEAUX PROGRAMMES

Jean-Noël Beury

INFORMATIQUE AVEC PYTHON

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

j'intègre

EXERCICES INCONTOURNABLES

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

2e édition

LES + EN LIGNE

DUNOD

Jean-Noël Beury

INFORMATIQUE AVEC PYTHON

MPSI·PCSI·PTSI
MP·PC·PSI·PT·TSI·TPC

j'intègre

EXERCICES INCONTOURNABLES

2e édition

DUNOD

Table des matières

Partie 13
Bases de données

Partie 14
Algorithmique numérique

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

https://dunod.com/EAN/9782100846238

Avant-propos

Cet ouvrage de la série « Exercices incontournables » traite de l'intégralité du nouveau programme d'informatique commune pour les deux années des différentes filières de classes préparatoires aux grandes écoles (sauf BCPST).

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

Dans les exercices de certains chapitres (« Prise en main de Python », « Terminaison, correction, complexité », « Matrices de pixels et images, traitement d'images », « Dictionnnaire, pile, file, deque », « Graphes », « Intelligence artificielle et jeux à deux joueurs », « Bases de données »), vous trouverez un rappel de cours détaillé présentant le vocabulaire utilisé.

Pour chaque exercice classique, vous trouverez :

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

Partie 1

Prise en main de Python

Plan

1

Prise en main de Python

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

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

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

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

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

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

```
| def uneFonction(n:int, X:[float], c:str, u) -> 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 et `c` une chaîne de caractères ; le type de `u` n'est pas précisé. Cette fonction renvoie une liste.

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

Analyse du problème

On utilise une boucle `for` pour parcourir les différents éléments de la liste. On peut alors calculer la somme des éléments de la liste pour en déduire la valeur moyenne. Une assertion est une aide de détection de bugs dans les programmes. La levée d'une assertion entraîne l'arrêt du programme.

Cours :
L'installation de Python 3 peut se faire avec Pyzo.

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

Une affectation se fait avec l'instruction `n=3` : *n* prend la valeur 3 (voir exercice 1.4 « Affectation, objet immuable, copie »).

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

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 `n` est `int`.

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

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

Les types de base des variables dans Python sont :

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

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

- Booléens : `bool`
  Les variables booléennes sont `True` (vrai) et `False` (faux).

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 nombres flottants (`float`) : `+`, `-`, `*`, `/`, `**`

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

Comparaisons :

`a==b` : cet opérateur compare *a* et *b*. Si *a=b*, Python retourne `True`, sinon `False`
`a!=b` : *a* différent de *b*
`a>b`, `a<b`, `a>=b`, `a<=b` : strictement supérieur, strictement inférieur, supérieur ou égal, inférieur ou égal

Opérations sur les booléens (`bool`) :

```
or         : ou
and        : et
not        : non
```

```
rep1=True              # type bool. 2 valeurs possibles : True ou False
a=12
b=10
rep2=(a==12)and(b==20) # rep2=False pour le test : a=12 et b=20
rep3=not(a==13)        # rep3=True, on pourrait écrire : rep3 = a!=13
rep4=(a==12)or(b==20)  # True pour le test : a=12 ou b=20
```

Les types de base des conteneurs dans Python sont :

- Chaînes de caractères : `str`
  Structure indicée immuable. On ne peut pas modifier les éléments d'une chaîne de caractères.
  `s="C'est une phrase"` : utilisation de guillemets "
  `s1='phrase'` : utilisation d'apostrophes '
  `n=len(s)` : n=6 (nombre de caractères)
  `s1[0]` : le premier caractère de `s1` a pour indice 0 : `'p'`
  `s1[n-1]` : dernier caractère : `'e'`
  `C1='ab'`
  `C2=C1*3` : retourne `'abcabcabc'`, répétition de `'ab'`
  La chaîne de caractères `C1` est concaténée 3 fois avec elle-même.
- Listes : `list`
  Structure de données muable. On peut modifier les éléments d'une liste.
  `L=[]` : crée une liste vide
  `L.append(3)` : ajoute 3 **à la fin de la liste** `L`. On obtient `L=[3]`
  `L.append(2)` : ajoute 2 à la fin de la liste `L`. On obtient `L=[3, 2]`
  `x=L.pop()` : **supprime le dernier élément** (ici 2) de la liste `L`
  On récupère cet élément dans la variable `x`
  `L3=['r', 3, 'te']` : crée la liste `L3` contenant des chaînes de caractères et des entiers
  `len(L3)` : affiche la **longueur de la liste** `L3`
  Pour extraire des éléments d'une liste, voir exercice 1.6 « Slicing, extraction de tranche ».
  `L=[2*i for i in range(5)]` : on obtient `[0, 2, 4, 6, 8]` (**création par compréhension**)
  `L=[2,3]*3` : **répétition** de la liste `[2, 3]`
  On a alors `L=[2, 3, 2, 3, 2, 3]`
  `L=L+[5,8]` : **concaténation** : `L=[2, 3, 2, 3, 2, 3, 5, 8]`

Remarque :

Les indices des listes contenant *n* éléments sont numérotés de 0 à *n*–1 dans Python (idem pour les tuples et les deques). Pour obtenir le premier élément de la liste :

```
a=L[0]       # la variable a prend la valeur 2
```

- Tuples : `tuple`
  Les tuples sont des structures indicées immuables. Une fois le tuple créé, il ne peut pas être modifié. On peut créer un tuple avec ou sans parenthèses.
  `M=(2, 3, 8)` : crée le tuple `M`
  Ne pas confondre avec les listes, où on met des crochets.
  On crée le même tuple si on omet les parenthèses

`n=len(M)` : n = nombre d'éléments du tuple `M`
`M=2, 3, 9`
`x=M[0]` : récupère dans la variable `x` l'élément d'indice 0 du tuple `M`
`a, b, c=M` : dépaquette un tuple
On récupère dans chaque variable un élément du tuple.
Il faut connaître à l'avance le nombre d'éléments du tuple.
`M2=M+(2, 5)` : concaténation de deux tuples : `M2=(2, 3, 9, 2, 5)`
`M3=(2, 1)*3` : création avec répétition : `M3=(2, 1, 2, 1, 2, 1)`

- Deques : `deque`
  Une deque (se prononce « dèque ») est une structure de données muable qui généralise le fonctionnement des piles et des files (voir exercice 11.6 « Utilisation des deques » dans le chapitre « Dictionnaire, pile, file, deque »). On peut ajouter et supprimer des éléments aux deux extrémités. 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(3)` : ajoute 3 à l'**extrémité droite** de `D`
  `D.appendleft(5)` : ajoute 5 à l'**extrémité gauche** de `D`
  `x=D.pop()` : supprime l'élément à l'**extrémité droite** de `D`
  `x=D.popleft()` : supprime l'élément à l'**extrémité gauche** de `D`
  `D1=deque([3, 8, 5])` : création de la deque `D1`

```
for elt in D1:     # affichage de tous les éléments de la deque D1
    print(elt)
```

- Dictionnaires : `dict`
  Pour l'utilisation des dictionnaires, voir les exercices 11.1 « Opérations de base sur les dictionnaires » et 11.2 « Comptage des éléments d'une liste à l'aide d'un dictionnaire » dans le chapitre « Dictionnaire, pile, file, deque ».

Remarque : Pour le type `None`, voir la remarque de la question 2 dans l'exercice 10.1 « Tri par insertion » dans le chapitre « Tris ». On ne l'utilisera pas dans les autres exercices.

Il existe deux catégories d'objets dans Python :

- les objets dont la valeur peut changer sont dits muables (en anglais : mutable) : listes, dictionnaires, deques (voir chapitre 11 « Dictionnaire, pile, file, deque »)… ;
- les objets dont la valeur ne peut pas changer sont dits immuables (en anglais : immutable) : entiers, nombres flottants, booléens, chaînes de caractères, tuples…

Voir les exercices 1.4 « Affectation, objet immuable, copie » et 1.5 « Passage par référence pour les listes, effet de bord » pour les affectations et les arguments d'entrée des fonctions.

Quelques fonctions intrinsèques :

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

Première utilisation de la boucle `for` :

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

Deuxième utilisation de la boucle `for` :

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

Boucle `while` :

```
i=0
while i<=10:         # ne pas oublier : à la fin de la ligne while
    print(i)         # affichage de i
    i=i+1            # on incrémente i de 1 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
```

L'instruction `break` fait sortir d'une boucle `while` ou `for` et passe à l'instruction suivante (voir exercice 10.6 « Tri à bulles » dans le chapitre « Tris »). Lorsqu'il y a plusieurs boucles imbriquées, l'instruction `break` ne fait sortir que de la boucle la plus interne.

Définition d'une fonction :

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

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

Structure conditionnelle :

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

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.

```
import math          # importation 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 p)…

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
```

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

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

```
m.sin(x)==0          # si x=pi, ceci retourne pourtant False
```

mais les instructions suivantes :

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

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)
```

Voir exercice 1.4 « Affectation, objet immuable, copie » pour l'utilisation du module `copy`.

On utilisera la bibliothèque `PIL` dans le chapitre 9 « Matrices de pixels et images, traitement d'images ».

1. On considère une liste non vide `L`. On suppose que les éléments de la liste sont des nombres entiers ou flottants. On définit une variable `S` permettant de calculer la somme des éléments de la liste.

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

Remarque :

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

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

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

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

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

2.

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

3.

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

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

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

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

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

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

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

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

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

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

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

4.

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

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

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

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

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

Si on exécute l'instruction `print('moyenne :',rec_moy2([3, 4.0, 'a']))`, Python affiche :

```
assert type(L[i])==float or type(L[i])==int
AssertionError
```

Remarques :

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

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

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

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

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

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

On peut écrire :

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

ou

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

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

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

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

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

Exercice 1.2 : Boucle, test, fonction (banque PT 2015)

On pourra utiliser `L.reverse()` qui permet d'inverser les éléments de la liste `L`.

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

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

La fonction vérifiera que `n` est un entier avec `assert`.

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

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

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

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

Analyse du problème

Cet exercice permet de s'entraîner à manipuler les fonctions, les boucles, les tests et les différents types rencontrés dans Python.

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

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

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

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

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

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

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

```
print('Décomposition en base 10 : ',calcul_base10(n))
```

L'instruction `assert expression` de Python vérifie la véracité d'une expression booléenne et interrompt brutalement l'exécution du programme si ce n'est pas le cas.

2.

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

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

3.

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

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

4.

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

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

Remarque :

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

- Dépaquetage d'un tuple :

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

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

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

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

- On définit un tuple *A* :

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

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

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

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

Exercice 1.3 : Variables locales, variables globales

On considère le programme suivant :

```
def f() :
    global b
    print('d =', d)
    print('Premier print dans la fonction f : b =', b)
    a=3
    c=5
    b=b+c
    print('Deuxième print dans la fonction f : b =', b)
    print('Troisième print dans la fonction f : a =', a)
   return  # on pourrait supprimer cette ligne
           # ou écrire return None

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

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

Analyse du problème

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

Cours :

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

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

Le programme Python affiche :

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

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

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

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

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

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

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

Exercice 1.4 : Affectation, objet immuable, copie

La fonction `deepcopy(L)` du module `copy` permet de réaliser une copie profonde de la liste `L`.

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

```
i=3
j=i
print('Avant modification de i : i, j =', i, j)
i=5
print('Après modification de i : i, j =', i, j)
```

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

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

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

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

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

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

Analyse du problème

Ce programme permet de comprendre les problèmes rencontrés lors de copies de listes, deques et dictionnaires (voir chapitre 11 « Dictionnaire, pile, file, deque »).

Cours :

Il existe deux catégories d'objets dans Python :

- les objets dont la valeur peut changer sont dits muables (en anglais : mutable) : listes, dictionnaires, deques… ;
- les objets dont la valeur ne peut pas changer sont dits immuables (en anglais : immutable) : entiers, nombres flottants, booléens, chaînes de caractères, tuples…

Objets muables – Partage de valeurs par plusieurs variables

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

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

- Création d'un objet 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]`.

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

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

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

```
L2=L1
```

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

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

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

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

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

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

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

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

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

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

Comme dans les problèmes de concours, on utilise le langage suivant : le terme « liste » appliqué à un objet Python signifie qu'il s'agit d'une variable de type `list`. Idem pour les autres types : `int`, `float`, `bool`, `str`, `tuple`, `dict`, `deque`…

Objets immuables (non modifiables)

```
a=10
```

Cette affectation réalise les opérations suivantes :

- Création d'un objet immuable (appelé `obj2`) de type `int` à une adresse mémoire. Cet objet possède un identifiant (adresse mémoire), un type et une valeur. La valeur de `obj2` vaut : `10`.

- Création de la variable `a`.
- Association de la variable `a` avec l'objet `obj2` contenant la valeur `10`.

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

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

Cours :

```
b=a
```

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

```
a=11
```

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

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

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

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

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

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

Cours :

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

- La fonction `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()` réalise une copie profonde pour les structures imbriquées. Si les éléments sont des listes, alors la copie profonde copie bien les listes imbriquées.

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

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

La copie superficielle s'applique également aux dictionnaires et deques (voir chapitre 11 « Dictionnaire, pile, file, deque »).

Remarque :

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

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

1. Python affiche :

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

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

2. Python affiche :

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

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

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

3. Python affiche :

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

L'instruction `L4=copy.copy(L3)` permet de réaliser une copie superficielle de `L3`. Les listes `L3` et `L4` ont des adresses mémoire différentes.

La modification d'un élément de `L4` n'a donc aucune conséquence sur `L3` puisque les éléments ne contiennent de structure imbriquée.

4. Python affiche :

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

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

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

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

Exercice 1.5 : Passage par référence pour les listes, effet de bord

On considère le programme suivant :

```
def f(a, b, L):
    a+=1
    print('Print dans la fonction : a =', a)
    b=b+1
    print('Print dans la fonction : b =', b)
    print('Print dans la fonction : d =', d)
    L.append(4)       # on ajoute un élément dans L
    print('Print dans la fonction : L =', L)
    return a

a=3
b=5
d=3
L=[1, 2, 3]
c=f(a, b, L)
print('Print après la fonction : a =', a)
print('Print après la fonction : b =', b)
print('Print après la fonction : c =', c)
print('Print après la fonction f : L =', L)
```

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

Analyse du problème

Ce programme permet de comprendre les problèmes rencontrés lors de l'utilisation de listes dans les arguments d'entrée des fonctions.

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

- 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.

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.

Python affiche :

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

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

Avant l'appel de la fonction `f`, la variable `b` fait référence à `5`. La variable `b` est passée par référence dans la fonction `f` et fait toujours référence à `6`. L'instruction `b=b+1` dans la fonction `f` ne peut affecter `5` qui est immuable. La variable `b` dans la fonction `f` fait donc référence à une nouvelle valeur `6`. La variable `b` du programme principal garde donc la même valeur `6`.

Pour simplifier, cela revient à dire que les variables `a` et `b` sont passées par valeur : l'exécution de la fonction `f` évalue d'abord `a` et `b` puis exécute `f` avec les valeurs calculées `a` et `b`.

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

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

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.

Remarque : Voir l'exercice précédent, « Affectation, objet immuable, copie », pour avoir une copie profonde avec la fonction `deepcopy()`.

Exercice 1.6 : Slicing, extraction de tranche

On considère le programme suivant :

```
L1=[i for i in range(2, 7, 2)]   # création par compréhension
print('L1 =', L1)
print(L1[:-1])
```

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

Analyse du problème
La technique du slicing, ou extraction de tranche, permet d'extraire des éléments d'une liste.

Cours :
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 ». On utilise la syntaxe :

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

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

Exemple pour `L=[3, 5, 1, 8, 10]` : `L[1:4]` renvoie `[5, 1, 8]`.

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

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

Cours :
On peut utiliser l'extraction de tranche avec un pas différent de 1. On utilise alors la syntaxe suivante :

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

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

Exemple pour `L=[3, 5, 1, 8, 10]` : `L[1:4:2]` renvoie `[5, 8]`.

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

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

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

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

`L[1:]` renvoie tous les éléments dont les indices sont compris entre 1 inclus et (indice du dernier élément de la liste + 1) exclu. Python retourne `[5, 1, 8, 10]`.

On peut créer une liste par compréhension :

```
L1=[3*i for i in range(3)]
```

On obtient : `L1=[0, 3, 6]`.

On peut également utiliser une boucle `for` :

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

On obtient : `L1=[0, 3, 6]`.

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

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

Cours :
On 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"
```

Python affiche :

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

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

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

Graphiques

2

Exercice 2.1 : Tracé d'une fonction avec matplotlib

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

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

Les fonctions suivantes permettent le tracé de fonctions :

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

Définir les deux fonctions `f1` et `f2` dans Python en utilisant le module `math`. Tracer les courbes représentatives des deux fonctions sur l'intervalle [0, 2] avec un pas de 0,05. Le graphique doit avoir les caractéristiques suivantes :

- Utilisation de listes.
- Courbe représentative de `f1` : épaisseur du trait égale à 3, couleur bleue.
- Courbe représentative de `f2` : points non reliés représentés par *, couleur rouge.
- Légender les deux courbes.
- Afficher `'x'` pour l'axe des abscisses et `'y'` pour l'axe des ordonnées.
- Afficher le titre : `'Tracé de fonctions'`.
- Axe des x compris entre 0 et 2, axe des y compris entre 0 et 1,5.

Analyse du problème

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

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

Cours :

Le module `matplotlib.pyplot` de la bibliothèque `matplotlib` permet d'afficher des graphiques. On l'importe à l'aide de la commande :

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

Il faut connaître quelques fonctions du module `matplotlib.pyplot` :

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

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

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

def f2(x):         # définition de la fonction f2
                   # argument d'entrée : float x
    return m.sin(x)+0.1


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

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

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


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

Exercice 2.2 : Tracé d'un histogramme avec matplotlib

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

Les fonctions suivantes permettent le tracé d'histogrammes :

```
import matplotlib.pyplot as plt  # 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 la figure à l'écran
```

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

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

Analyse du problème

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

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

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

```
import matplotlib.pyplot as plt
  # 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 la figure à l'écran
```

Partie 2

Terminaison, correction, complexité

Plan

Terminaison, correction, complexité

3

Exercice 3.1 : Comparaison de deux listes (Mines Ponts 2017)

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

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

3. Préciser le type de retour de cette fonction.

Analyse du problème

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

Cours :
Terminaison d'un programme

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

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

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

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

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

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

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

Voir chapitre 6 « Récursivité » pour des exemples avec des fonctions récursives.

Correction d'un programme

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

Rédaction pour un programme itératif

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

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

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

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

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

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

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

On a démontré que le programme est correct.

Rédaction pour un programme récursif

Voir chapitre 6 « Récursivité » pour la rédaction avec des fonctions récursives.

On distingue parfois correction partielle et correction totale :

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

Complexité

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

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

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

```
def rec_max(L):
    # la fonction renvoie le maximum des éléments de la liste 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 en $O(1)$ (ne dépend pas de n),
- logarithmique en $O(\log n)$,
- linéaire en $O(n)$,
- quasi linéaire en $O(n \log n)$,
- quadratique en $O(n^2)$,
- polynomiale en $O(n^p)$,
- exponentielle en $O(2^n)$.

1.

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

```
    return True

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

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

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

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

On a donc $3 + 3n$ 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.

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

Exercice 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 σ^2 les estimateurs non biaisés de l'espérance et de la variance, n le nombre d'éléments $H_1, H_2, ..., H_n$. On définit alors :

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

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

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

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

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

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

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

Analyse du problème

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

1.

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

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

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

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

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

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

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

Exercice 3.3 : Décomposition en base *b* d'un entier (CCP MP Maths 2015)

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

2. On considère la fonction `mystere` suivante :

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

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

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

k	1	2	…
c_k			…
t_k			…
n_k			…

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

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

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

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

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

Analyse du problème

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

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

Remarque :

On peut effectuer les divisions euclidiennes suivantes :

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

$21//2 = 10$ = quotient

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

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

b) Division euclidienne du quotient précédent par 2 :

$10//2 = 5$

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

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

c) Division euclidienne du quotient précédent par 2 :

$5//2 = 2$

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

d) Division euclidienne du quotient précédent par 2 :

$2//2 = 1$

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

e) Division euclidienne du quotient précédent par 2 :

$1//2 = 0$

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

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

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

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

2.

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

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

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

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

3.

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

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

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

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

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

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

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

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

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

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

5.

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

Remarque :

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

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

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

Exercice 3.4 : Recherche du nombre de zéros (banque PT 2015)

Soit un entier naturel *n* non nul et une liste `L` de longueur *n* dont les termes valent 0 ou 1. On cherche le nombre maximal de 0 contigus dans `L` (c'est-à-

dire figurant dans des cases consécutives). Par exemple, le nombre maximal de zéros contigus de la liste `L1` suivante vaut 4 :

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
L1[i]	0	1	1	1	0	0	0	1	0	1	1	0	0	0	0

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

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

Par exemple, les appels `nombreZeros(L1,4)`, `nombreZeros(L1,1)` et `nombreZeros(L1,8)` renvoient respectivement les valeurs 3, 0 et 1.

2. Comment obtenir le nombre maximal de zéros contigus d'une liste `L` connaissant la liste des `nombreZeros(L,i)` pour $0 \leq i \leq n-1$?

En déduire une fonction `nombreZerosMax(L)`, de paramètre `L`, renvoyant le nombre maximal de 0 contigus d'une liste `L` non vide. On utilisera la fonction `nombreZeros`.

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

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

Analyse du problème

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

1.

```
def nombreZeros(L, i):
    # la fonction renvoie nbr_zeros pour la liste L
    # et un indice i (int)
    if L[i]==1:
        return 0
    else:
        nbr_zeros=0
        j=i
        while j<len(L) and L[j]==0:
            # il faut mettre j<len(L) avant L[j]
            # sinon message d'erreur pour j=len(L)
            nbr_zeros+=1   # incrémente de 1
                           # le nombre de zéros
            j+=1
    return nbr_zeros
```

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

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

```
def nombreZerosMax(L):
    # la fonction renvoie le nombre maximal de 0 contigus
    # de la liste L
    Z=[]                              # initialisation de la liste
    for i in range(len(L)):
        Z.append(nombreZeros(L,i)) # stocke nombreZeros(L,i)
    max_Z=Z[0]                        # recherche du maximum
                                      # de la liste Z
    for i in range(1,len(Z)):         # i varie entre 1 inclus
                                      # et len(Z) exclu
        if Z[i]>max_Z:
            max_Z=Z[i]
    return max_Z
print (nombreZeros(L1, 13))
print(nombreZerosMax(L1))
```

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

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

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

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

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

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

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

Si `valeur` est non nul, il est alors inutile d'appeler la fonction `nombreZeros(L, i+1)` mais on appelle `nombreZeros(L, i+valeur+1)`. Entre i et i + `valeur`, la liste `L` ne contient que des

zéros et `L[i+valeur+1]` vaut nécessairement 1 puisque la fonction `nombreZeros` donne le nombre de zéros consécutifs, sauf si on est en fin de liste.

```
def nombreZerosMax2(L):
    # la fonction renvoie le nombre maximal de 0 contigus
    # de la liste L
    valeur_max=0
    while i<len(L):   # i ne doit pas dépasser len(L)-1
        valeur=nombreZeros(L,i)
        if valeur==0:
            i+=1
        else:
            i+=valeur+1
            if valeur>valeur_max:
                valeur_max=valeur
    return valeur_max
```

Remarque : On utilisera cette même technique d'optimisation dans l'exercice 4.3 « Recherche d'un mot dans un texte, boucles imbriquées » dans le chapitre « Algorithmes ».

Partie 3

Algorithmes de recherche

Plan

Algorithmes

4

Exercice 4.1 : Recherche du minimum et du maximum d'une liste de nombres, complexité

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

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

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

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

Analyse du problème

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

1.

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

Remarques :

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

2.

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

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

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

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

1 affectation pour `maxi` : 1 opération élémentaire.

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

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

Le nombre d'opérations élémentaires vaut 2.

On a n itérations. Le nombre d'opérations élémentaires vaut $1+2n$.

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

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

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

Exercice 4.2 : Assertion. Recherche du maximum, du second maximum d'une liste

On considère la liste : `L=[9, 10, 56, 28, -8]`.

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

2. Écrire le programme principal permettant d'afficher le maximum et le second maximum de la liste `L`.

Analyse du problème

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

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

1.

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

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

2.

```
L=[9, 10, 56, 28, -8]
max1, max2=rec_max2(L)
print('Maximum de L :', max1)
print('Second maximum de L :', max2)
```

Exercice 4.3 : Recherche d'un mot dans un texte, boucles imbriquées

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

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

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

Analyse du problème

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

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

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

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

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

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

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

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

2.

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

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

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

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

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

Algorithmes de dichotomie

5

Exercice 5.1 : Recherche d'un élément dans une liste non triée, algorithme naïf, complexité

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

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

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

3. Écrire une fonction `rec_elt2` 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`.

Analyse du problème

On parcourt la liste en partant du premier élément jusqu'à ce qu'on trouve `x`. On étudiera dans l'exercice suivant la méthode dichotomique, qui ne s'applique qu'aux listes triées.

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

```
def rec_elt1(L, x):
    # la fonction retourne True si x est dans la liste L,
    # False sinon
    n=len(L)              # nombre d'éléments de la liste
    for i in range(n):    # i varie entre 0 inclus et n exclu
        if x==L[i]:       # teste si x=L[i]
            return True
            # return provoque un arrêt de la boucle
            # et une sortie de la fonction
    return False
```

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

On cherche à calculer le nombre d'opérations élémentaires :

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

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

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

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

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

On utilise un indice *i* pour repérer la position dans la liste. Les indices des éléments d'une liste commencent à 0 avec Python.

```
def rec_elt2(L, x):
    # la fonction retourne True, i si x est dans
    # la liste L à l'indice i, sinon retourne False, -1
    n=len(L)   # nombre d'éléments de la liste
    i=0
    while i<n:
        if L[i]==x:
            return True, i
            # return provoque un arrêt de la boucle
            # et une sortie de la fonction
        i+=1
    return False, -1
```

Exercice 5.2 : Recherche dichotomique dans une liste triée, complexité

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

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

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

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

Analyse du problème

La méthode dichotomique utilise le fait que la liste est triée. Elle consiste à comparer l'élément recherché à l'élément se trouvant au milieu d'une liste triée. Comme la liste est triée, cela permet d'éliminer une 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…

Voir exercice 6.5 « Recherche dichotomique dans une liste triée, version récursive » dans le chapitre « Récursivité » pour une version récursive de ce programme.

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

On verra la méthode générale « diviser pour régner » qui profite de la subdivision pour effectuer moins de calculs : voir l'exercice 6.2 « Exponentiation naïve, exponentiation rapide » dans le chapitre « Récursivité », et les exercices 10.3 « Tri rapide » et 10.4 « Tri par partition-fusion » dans le chapitre «Tris ».

1.

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

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

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

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

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

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

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

3 affectations, 1 calcul : 4 opérations élémentaires

a. Calcul du nombre d'opérations élémentaires à chaque itération :

- Trois comparaisons : 3 opérations élémentaires.
- Calcul du milieu et une affectation : 2 opérations élémentaires.
- Appel de l'élément `L[milieu]` et 1 test : 2 opérations élémentaires.
- Une affectation : 1 opération élémentaire.

Le nombre d'opérations élémentaires vaut 10.

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

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

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

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

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

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

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

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

Remarque :

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

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

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

Partie 4

Récursivité

Plan

Récursivité

6

Exercice 6.1 : Factorielle d'un entier naturel

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

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

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

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

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

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

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

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

Analyse du problème

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

Cours :

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

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

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

1.

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

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

2.

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

Cours :

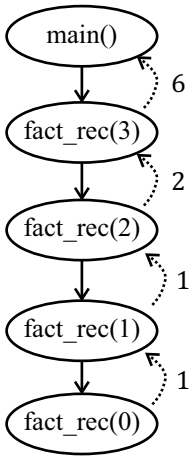
Pour démontrer la terminaison d'un programme, on cherche une grandeur positive, que l'on appelle variant de boucle, qui décroît entre deux appels de la fonction récursive et qui converge vers une valeur d'un cas correspondant à un appel de la condition d'arrêt.

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

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

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

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



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

Remarque : Dans l'exercice 10.4 « Tri par partition-fusion » dans le chapitre « Tris », l'arbre des appels de la fonction récursive ressemble encore plus à un « arbre » !

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

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

Remarque :

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

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

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

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

Cours :

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

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

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

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

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

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

Remarque :

On distingue parfois correction partielle et correction totale :

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

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

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

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

Remarque :

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

La complexité d'une fonction récursive est souvent beaucoup plus importante (voir exercice 6.7 « Suite des nombres de Fibonacci ») que la complexité de la fonction itérative.

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

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

Exercice 6.2 : Exponentiation naïve, exponentiation rapide

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

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

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

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

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

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

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

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

Évaluer la complexité de la fonction `puiss_rapide`.

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

Analyse du problème

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

1.

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

2.

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

Remarque :

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

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

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

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

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

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

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

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

Remarque :

On distingue parfois correction partielle et correction totale :

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

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

À chaque appel de la fonction récursive, on a :

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

On en déduit la relation de récurrence :

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

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

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

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

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

À chaque appel de la fonction récursive, on a :

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

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

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

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

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

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

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

Finalement, on obtient :

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

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

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

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

D'après l'énoncé, on a :

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

La complexité est linéaire en $O(n)$.

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

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

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

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

À chaque appel de la fonction récursive, on a :

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

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

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

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

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

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

Par exemple pour 2^4, on a :

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

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

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

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

Explications du terme « exponentiation rapide » :

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

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

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

Remarques :

- La méthode « diviser pour régner » profite de la subdivision pour effectuer moins de calculs (voir les exercices 10.3 « Tri rapide » et 10.4 « Tri par partition-fusion » dans le chapitre «Tris »).
- La méthode dichotomique divise uniquement le problème initial et élimine une partie des données (voir exercice 5.2 « Recherche dichotomique dans une liste triée, complexité » dans le chapitre « Algorithmes de dichotomie »).

Exercice 6.3 : Tours de Hanoï

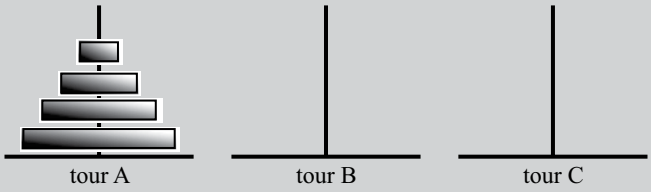
On considère n disques de diamètre différent empilés par ordre décroissant sur une tour de départ (tour A sur la figure ci-dessous). Les deux autres tours n'ont pas de disque. L'objectif est de déplacer les disques de la tour A (tour de départ) vers la tour C (tour d'arrivée) en utilisant les deux règles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

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

On considère la suite définie par récurrence : $u_{n+1} = au_n + b$. On pose : $r = \frac{b}{1-a}$.
On admet que : $u_n = a^n\left(u_0 - r\right) + r$.

On définit la liste `tour` : `tour[0]` est une liste représentant les disques de la tour A, `tour[1]` (respectivement `tour[2]`) représente les disques de la tour B (respectivement tour C).

Par exemple, pour $n = 4$, on a : `tour = [[4, 3, 2, 1], [], []]`.



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

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

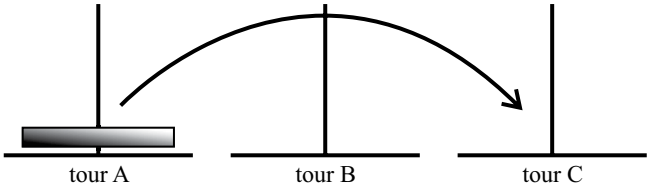
3. Écrire une fonction récursive `hanoi(tour, n, a, b, c)` qui admet comme arguments `tour` la liste décrite précédemment, `n` le nombre de disques, `a` la tour de départ, `b` la tour intermédiaire et `c` la tour d'arrivée. Dans l'exemple précédent, la fonction `hanoi(tour, 4, 0, 1, 2)` doit retourner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

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

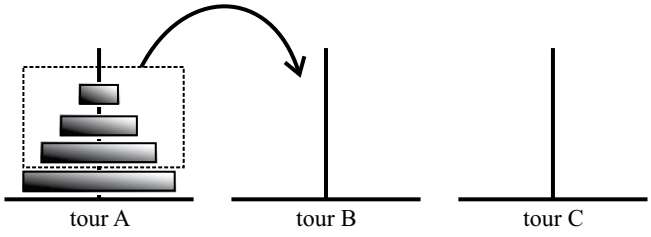
Analyse du problème

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

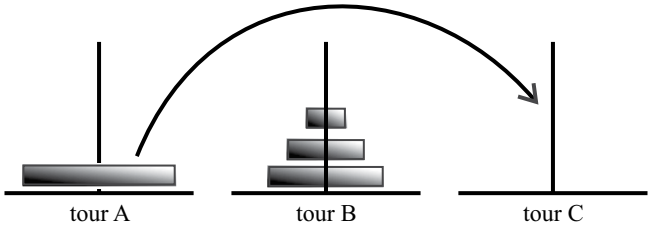
1. La résolution du problème est évidente pour $n = 1$ puisqu'il suffit de dépiler le disque de la tour A et de l'empiler dans la tour C.



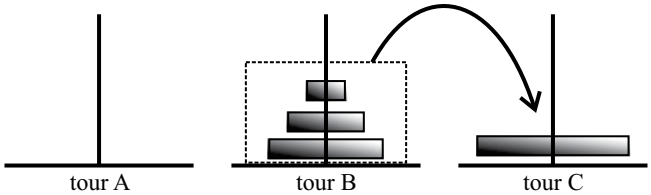
2. On souhaite déplacer n disques de la tour A vers la tour C. On considère $n = 4$ dans l'exemple suivant.



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



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



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

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

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

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

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

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

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

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

À chaque appel de la fonction récursive, on a :

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

On en déduit la relation de récurrence :

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

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

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

On en déduit que :

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

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

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

Exercice 6.4 : Recherche du pgcd (CCP MP Maths 2016)

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

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

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

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

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

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

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

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

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

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

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

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

Analyse du problème

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

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

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

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

2.

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

Cours :

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

3.

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

Remarque :

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

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

Appel de `euclide_rec(12,4)`

Appel de `euclide_rec(4,0)`

Retour de 4

Retour de 4

4. a.

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

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

b. La suite de Fibonacci est définie par : $F_{n+2} = F_{n+1} + F_n$ et $0 \leq F_n < F_{n+1}$.
On en déduit que le reste de la division euclidienne de F_{n+2} par F_{n+1} est F_n.
D'après la question précédente :

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

On obtient alors la suite de valeurs :

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

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

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

```
def fibo1(n):
    # renvoie le nombre de Fibonacci Fn pour l'entier naturel n
    if n==0:
        return 0
```

```
    elif n==1:
        return 1
    else:
        F=[]
        F.append(0)
        F.append(1)
        for i in range(2, n+1):
            F.append(F[i-1]+F[i-2])
        return F[i]
```

b) Deuxième version, sans utiliser de liste.

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

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

c) Troisième version

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

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

6.

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

```
    else:
        return (fibo_rec(n-1)+fibo_rec(n-2))
                    # appel récursif
```

Remarque : Ceci sera traité dans l'exercice 14.1 « Suite des nombres de Fibonacci, Top Down et Bottom Up » dans le chapitre « Programmation dynamique » pour une optimisation du programme récursif.

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

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

Exercice 6.5 : Recherche dichotomique dans une liste triée, version récursive

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

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

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

Analyse du problème

On a étudié une version itérative de ce programme (voir exercice 5.2 « Recherche dichotomique dans une liste triée, complexité » dans le chapitre « Algorithmes de dichotomie »).

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

1.

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

```
            # appel récursif
        else:
            # recherche entre début de liste et milieu-1
            return rec_dicho_recursive(L[:milieu], x)
            # appel récursif

L=[9, 10, 11, 15, 16, 17, 18, 25, 32]
x=32
print(rec_dicho_recursive(L,x))
```

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

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

À chaque appel de la fonction récursive, on a :

- un test pour savoir si la liste est vide ;
- un calcul du milieu ;
- un test pour savoir si l'élément est trouvé ;
- un appel de la fonction récursive avec $n/2$.

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

$$T(n) = T\left(\frac{n}{2}\right) + 3 \text{ avec } T(1) = 1$$

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

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

Il ne reste plus qu'un seul appel de la fonction récursive quand $\frac{n}{2^k} = 1$, soit $\ln(n) = k \ln(2)$.

Finalement, on obtient :

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

On appelle $k+1$ fois la fonction récursive. On a donc $T(n = 2^k) = 1 + 3(k+1) = 4 + 3\log_2(n)$.

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

Remarque :

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

On a la même complexité que pour la version itérative.

Exercice 6.6: Dessins de fractales

On utilise le module `turtle` pour le tracé de fractales. La tortue trace un trait le long du chemin parcouru.

`from turtle import *` : fenêtre graphique `turtle`. La tortue est le triangle affiché au centre de la fenêtre de coordonnées (0,0)

`reset()` : efface l'écran

`forward(150)` : avance la tortue de 150 pixels

`penup()` : lève le crayon et permet ensuite de déplacer la tortue (avec `forward`) sans tracer

`pendown()` : pose le crayon et permet ensuite de déplacer la tortue (avec `forward`) avec un trait

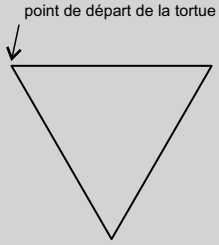
`goto(120,200)` : déplace la tortue au point de coordonnées (120, 200)

`left(60)` : fait tourner la tortue vers la gauche d'un angle de 60° sans avancer

`right(60)` : fait tourner la tortue vers la droite d'un angle de 60° sans avancer

`mainloop()` : laisse la fenêtre graphique `turtle` ouverte à la fin du programme

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



2. Écrire une fonction `polygone` qui admet comme arguments deux entiers naturels `n` et `a`. Cette fonction trace un polygone régulier à `n` côtés de même longueur `a`.

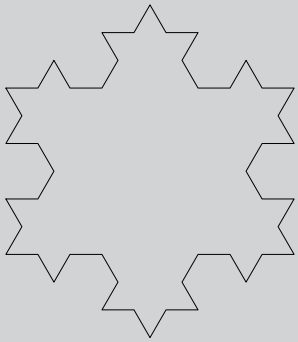
3. On souhaite tracer le segment de Koch de longueur *a* à l'ordre *n*.

- On part d'un segment de longueur *a* à l'ordre 0.
- À l'étape 1, on remplace le tiers du segment central par un triangle équilatéral sans base au-dessus.
- On réitère le processus *n* fois.

ordre 0 ordre 1 ordre 2

Écrire une fonction récursive `koch` qui admet comme arguments deux entiers naturels `n` et `a`. Cette fonction trace un segment de Koch de longueur *a* à l'ordre *n*.

4. Pour tracer le flocon de neige, on part d'un triangle équilatéral et on applique la fonction `koch` à l'ordre *n* à chacun des côtés du triangle équilatéral de longueur *a*. La figure représente le flocon de Von Koch à l'ordre 2.



Écrire le programme principal permettant de tracer un flocon de Von Koch à l'ordre *n*.

Analyse du problème

Le module graphique `turtle` permet de piloter une tortue afin de tracer des figures géométriques.

Une figure fractale est un objet géométrique « infiniment morcelé » dont les détails sont observables à une échelle arbitrairement choisie. Le flocon de Von Koch est un exemple de courbe fractale. En zoomant sur une partie de la figure, on retrouve toute la figure : on dit qu'elle est autosimilaire. À chaque étape, la longueur de la base est multipliée par $\left(\frac{4}{3}\right)^n$. Le périmètre du flocon à l'étape *n* est : $3a\left(\frac{4}{3}\right)^n$ et tend vers l'infini si *n* tend vers l'infini. La courbe fractale n'admet de tangente en aucun point.

1. Pour tracer le triangle équilatéral de côté *a* :

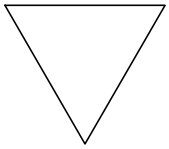
- On avance de *a*/3.
- On tourne la tortue vers la droite de 120°.
- On avance de *a*/3.
- On tourne la tortue vers la droite de 120°.
- On avance de *a*/3.
- On tourne la tortue vers la droite de 120°.

La tortue revient à sa position initiale avec le même angle.

```
from turtle import *  # bibliothèque pour la fenêtre graphique
                      # turtle
def triangle(a): # la fonction dessine un triangle équilatéral
                 # de côté a
```

```
    forward(a)    # avance la tortue de a pixels
    right(120)    # tourne la tortue vers la droite de 120°
    forward(a)    # avance la tortue de a pixels
    right(120)    # tourne la tortue vers la droite de 120°
    forward(a)    # avance la tortue de a pixels
    right(120)    # tourne la tortue vers la droite de 120°
triangle(100)     # appel de la fonction triangle avec un côté
                  # de 100 pixels
mainloop()        # laisse la fenêtre graphique turtle ouverte
                  # à la fin du programme
```

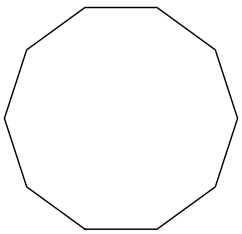
On obtient la figure suivante :



2.

```
def polygone(n, a):     # la fonction dessine un polygone
                        # régulier à n côtés de longueur a
    for i in range(n): # i varie entre 0 inclus et n exclu
        forward(a)      # avance la tortue de a pixels
        left(360/n)     # tourne la tortue vers la gauche de
                        # 360/n degrés
polygone(10, 50) :      # 10 côtés de longueur 50
mainloop()              # laisse la fenêtre graphique turtle
                        # ouverte à la fin du programme
```

On obtient la figure suivante :



3. On considère un segment de Koch à l'ordre 0 de longueur a. On décompose ce segment en trois segments de longueur $a/3$.

- On avance de $a/3$.
- On tourne la tortue vers la gauche de 60°, on avance de $a/3$.
- On tourne la tortue vers la droite de 120°, on avance de $a/3$.
- On tourne la tortue vers la gauche de 60° et on avance de $a/3$.

On obtient le segment de Koch à l'ordre 1 :

ordre 0 ordre 1

Pour tracer le segment de Koch de longueur *a* à l'ordre *n* :

- On appelle la fonction `koch(n-1, a/3)` permettant de tracer le segment de Koch à l'ordre *n*–1 de longueur *a*/3.
- On tourne la tortue vers la gauche de 60°. On appelle la fonction `koch(n-1, a/3)` permettant de tracer le segment de Koch à l'ordre *n*–1 de longueur *a*/3.
- On tourne la tortue vers la droite de 120°. On appelle la fonction `koch(n-1, a/3)` permettant de tracer le segment de Koch à l'ordre *n*–1 de longueur *a*/3.
- On tourne la tortue vers la gauche de 60°. On appelle la fonction `koch(n-1, a/3)` permettant de tracer le segment de Koch à l'ordre *n*–1 de longueur *a*/3.

La fonction `koch(n-1, a/3)` trace le segment de Koch en faisant appel 4 fois à la fonction `koch` à l'ordre *n*–2. Cette fonction `koch` à l'ordre *n*–2 fait appel à la fonction `koch` à l'ordre *n*–3...

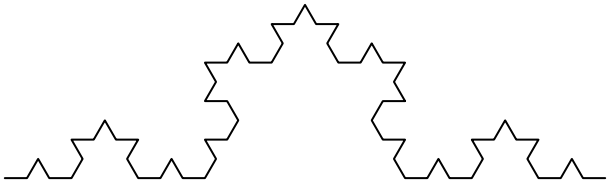
On peut construire le segment de Koch à l'ordre 1 à partir de quatre segments de Koch à l'ordre 0 :

- On appelle la fonction `koch(0, a/3)` permettant de tracer le segment de Koch à l'ordre 0 de longueur *a*/3.
- On tourne la tortue vers la gauche de 60°. On appelle la fonction `koch(0, a/3)` permettant de tracer le segment de Koch à l'ordre 0 de longueur *a*/3.
- On tourne la tortue vers la droite de 120°. On appelle la fonction `koch(0, a/3)` permettant de tracer le segment de Koch à l'ordre 0 de longueur *a*/3.
- On tourne la tortue vers la gauche de 60°. On appelle la fonction `koch(0, a/3)` permettant de tracer le segment de Koch à l'ordre 0 de longueur *a*/3.

Il y a bien une condition d'arrêt à la fonction récursive. Si $n = 0$, on avance la tortue de *a* pixels pour la fonction `koch(0, a)`.

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

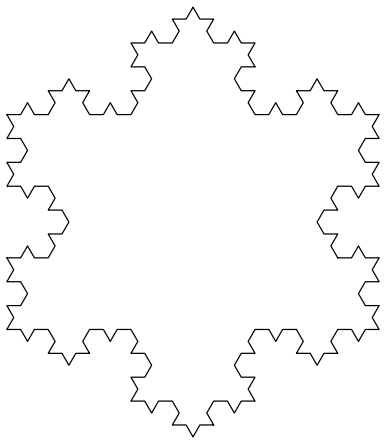
```
        # à l'ordre n-1
        koch(n-1, a/3) # appel récursif ordre n-1 et longueur a/3
        left(60)       # tourne la tortue vers la gauche de 60°
        # deuxième tiers : on appelle la fonction récursive
        # à l'ordre n-1
        koch(n-1, a/3) # appel récursif ordre n-1 et longueur a/3
        right(120)     # tourne la tortue vers la droite de 120°
        # troisieme tiers : on appelle la fonction récursive
        # à l'ordre n-1
        koch(n-1, a/3) # appel récursif ordre n-1 et longueur a/3
        left(60)       # tourne la tortue vers la gauche de 60°
        koch(n-1, a/3) # appel récursif ordre n-1 et longueur a/3
```



4. Il suffit d'appliquer la fonction `koch` à l'ordre *n* pour chaque côté du triangle équilatéral.

```
n=3          # flocon de Von Koch à l'ordre 3
a=300        # segment de longueur 300
koch(n, a)   # segment de Koch à l'ordre n
right(120)   # tourne la tortue vers la droite de 120°
koch(n, a)   # segment de Koch à l'ordre n
right(120)   # tourne la tortue vers la droite de 120°
koch(n, a)   # segment de Koch à l'ordre n
right(120)   # tourne la tortue vers la droite de 120°
mainloop()   # laisse la fenêtre graphique turtle ouverte
             # à la fin du programme
```

On obtient une courbe fermée :



Exercice 6.7 : Suite des nombres de Fibonacci

On note $(F_n)_{n\in\mathbb{N}}$ la suite des nombres de Fibonacci définie par : $F_0 = 0$, $F_1 = 1$, $\forall n \in \mathbb{N}$, $F_{n+2} = F_{n+1} + F_n$.

1. Écrire une fonction récursive `fibo1` qui permet de renvoyer le nombre de Fibonacci F_n. L'algorithme utilise-t-il la méthode « diviser pour régner » ?

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

Analyse du problème

La méthode « diviser pour régner » permet de décomposer le problème initial en deux sous-problèmes.

Afin d'éviter de calculer plusieurs fois le même nombre de Fibonacci, on utilise la technique de mémoïsation.

Cours :

La méthode « diviser pour régner » peut se décomposer en trois étapes :

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

Cette méthode donne de très bons résultats dans de nombreux problèmes : dichotomie, tri par partition-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 14 « Programmation dynamique »).

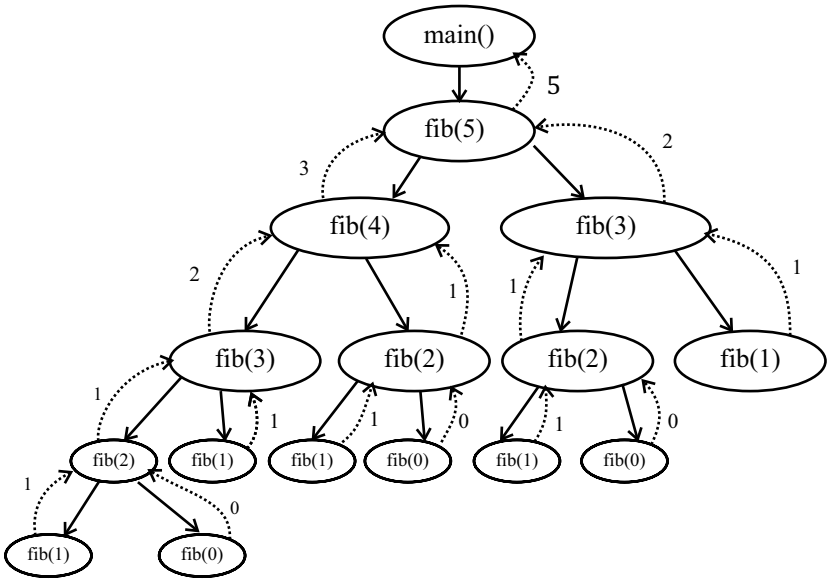
1.

```
def fibo1(n):
    # la fonction renvoie le nombre de Fibonacci Fn
    # pour l'entier n
    if n==0:
        return 0 # condition d'arrêt
    elif n==1:
        return 1 # condition d'arrêt
    else:
        return (fibo1(n-1)+fibo1(n-2))
                 # appel récursif
```

L'algorithme est de type « diviser pour régner » puisqu'on décompose le problème (calcul de `fibo1(n)`) en deux sous-problèmes (calcul de

`fibo1(n-1)` et `fibo1(n-2)`). On calcule récursivement chacun des deux sous-problèmes.

2. L'arbre ci-dessous représente les différents appels de la fonction `fibo1` que l'on note `fib`.



F_2 est recalculé 3 fois. F_3 est recalculé 2 fois.

L'inconvénient est que l'on augmente considérablement la complexité puisqu'on recalcule plusieurs fois la même valeur F_n.

Partie 5

Algorithmes gloutons

Plan

Algorithmes gloutons (sauf TSI et TPC)

7

Exercice 7.1 : Rendu de monnaie

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 cherche à rendre une certaine somme entière X en utilisant le moins de pièces, qui peuvent être identiques.

1. On utilise la méthode la plus intuitive qui consiste à commencer par rendre la plus grande pièce possible. Pour $X = 11$, on commence par rendre la pièce de 10.

On appelle $L[x]$ le nombre de pièces nécessaires pour rendre la somme x. La récurrence (1) peut s'écrire :

- $L[0] = 0$
- Si $x \geq 1$: $L[x] = 1 + L\left[x - S\left[i\right]\right]$ avec i le plus grand tel que $S\left[i\right] \leq x$.

Écrire une fonction récursive `rendu1` qui admet comme arguments une liste `S` et un entier `X`. La fonction retourne le nombre de pièces nécessaires pour rendre la somme `X` en utilisant la récurrence (1).

2. L'algorithme utilise-t-il la méthode « diviser pour régner » ? Pourquoi cette méthode est-elle appelée gloutonne ? Est-ce que `rendu1([1, 4, 6], 8)` retourne la solution optimale ?

Analyse du problème

On étudie plusieurs algorithmes permettant d'optimiser le rendu de monnaie. La méthode « diviser pour régner » permet de décomposer le problème initial en deux sous-problèmes.

La programmation dynamique (voir chapitre 14 « Programmation dynamique ») permet d'obtenir une solution optimale. On verra la différence entre la méthode gloutonne et la programmation dynamique.

1.

```
def rendu1(S, X):
    # la fonction renvoie le nombre de pièces nécessaires pour
    # rendre la somme X en utilisant la liste S :
    # S[i]=valeur de la pièce d'indice i
    if X==0:  # condition d'arrêt
        return 0
    else:
        # recherche de i le plus grand tel que S[i] <= X
        i=len(S)-1
        while S[i]>X:
            i=i-1
        # ajoute 1 au nombre de pièces
        # puisqu'on utilise la pièce S[i]
        # il reste donc à rendre la monnaie à X - S[i]
        return 1+rendu1(S, X-S[i]) # appel récursif
S=[1, 4, 6]
X=8
print(rendu1(S, X))  # on obtient : 3
```

Cours :

La méthode « diviser pour régner » peut se décomposer en trois étapes :

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

Dans la méthode gloutonne (greedy, en anglais), on effectue une succession de choix, chacun d'eux semble être le meilleur sur le moment. On résout alors le sous-problème mais on ne revient jamais sur le choix déjà effectué.

2. L'algorithme est de type « diviser pour régner » puisqu'on décompose le problème (calcul de $L[X]$) en un sous-problème (calcul de $L[X - S[i]]$ avec i le plus grand tel que $S[i] \leq X$). On calcule récursivement le sous-problème.

À chaque étape de l'algorithme, on commence par rendre la plus grande pièce possible, c'est-à-dire la plus grande pièce dont la valeur est inférieure à la somme à rendre. C'est la solution qui semble être la meilleure et la plus intuitive. On déduit alors de cette pièce la somme à rendre et on est ramené à un sous-problème avec une somme à rendre plus petite. On recommence jusqu'à obtenir une somme nulle.

Cet algorithme est très simple mais à chaque étape on n'étudie pas tous les cas possibles puisqu'on se contente de choisir la pièce la plus grande que l'on peut rendre.

Dans le cas où $S = [1, 4, 6]$ et $X = 8$, on n'obtient pas la solution optimale. L'algorithme glouton (greedy algorithm, en anglais) renvoie 3 (1 pièce de 6 et 2 pièces de 1) alors que la solution optimale est 2 (2 pièces de 4).

Exercice 7.2 : Problème du sac à dos

On considère un sac à dos dont la masse maximale est notée M. On cherche à maximiser la valeur totale des objets insérés dans le sac à dos. On dispose de n objets modélisés par la liste de listes `S` :

- `S[i][0]` désigne la valeur de l'objet d'indice i notée v_i (i varie de 0 à n–1).
- `S[i][1]` désigne la masse de l'objet d'indice i notée m_i (i varie de 0 à n–1).

On suppose dans tout le problème que $\sum_{i=0}^{n-1} m_i > M$ et que les masses sont des entiers. Le premier objet de la liste `S` a pour indice 0. La liste `S` est triée par ordre décroissant du rapport valeur/masse.

1. On utilise la méthode intuitive consistant à insérer au fur et à mesure les objets qui ont le plus grand rapport valeur/masse.
Écrire une fonction itérative `algo1` qui admet comme arguments une liste `S` et un entier `M`. La fonction retourne la valeur des objets que l'on peut insérer dans le sac à dos.

2. Pourquoi cette méthode est-elle appelée gloutonne ? Est-ce que `algo1 ([[15, 6], [60, 25], [10, 5], [7, 8], [10, 20]], 30)` retourne la solution optimale ?

Analyse du problème

On étudie plusieurs méthodes permettant de maximiser la valeur des objets insérés dans un sac à dos. La programmation dynamique (voir chapitre 14 « Programmation dynamique ») permet d'obtenir une solution optimale en utilisant deux techniques : « Top Down » et « Bottom Up ».

1.

```
def algo1(S, M):
    # S est une liste de listes avec [valeur, masse].
    # Les objets sont triés par ordre décroissant valeur/masse.
    # La fonction retourne la valeur des objets que l'on peut
    # insérer avec une masse maximale M
    v_total=0    # initialisation de la valeur totale des objets
    m_total=0    # initialisation de la masse totale des objets
    n=len(S)
    for i in range(n):
        if m_total+S[i][1]<=M: # teste si nouvelle masse totale<=M
            v_total+=S[i][0]   # calcule la nouvelle valeur totale
            m_total+=S[i][1]   # calcule la nouvelle masse totale
    return v_total  # retourne la valeur totale des objets
```

Avant d'insérer un objet, il faut tester que la nouvelle masse totale ne dépasse pas M.

2. Cette méthode est appelée méthode gloutonne car elle consiste à faire le meilleur choix sur le moment, c'est-à-dire insérer l'objet qui a le plus grand rapport valeur/masse.

```
M=30
S=[[15, 6], [60, 25], [10, 5], [7, 8], [10, 20]] # [valeur, masse]
print('ALGO1 :', algo1(S, M))                     # affiche 32
```

On peut calculer le rapport valeur/masse pour chaque objet. On obtient alors la liste suivante, qui est bien triée par ordre décroissant : [2.5, 2.4, 2.0, 0.875, 0.5].

- On insère le premier objet de valeur 15 et de masse 6.
- On ne peut pas insérer le deuxième objet de masse 25 car la masse totale 6+25 dépasse 30.
- On insère le troisième objet de valeur 10.
- On insère le quatrième objet de valeur 7.

On obtient une valeur totale 32 dans le sac à dos.

Cette méthode ne donne pas toujours la valeur optimale. Dans ce cas particulier, `algo1` renvoie 32 alors que la solution optimale est 70. On utilisera la programmation dynamique (voir chapitre 14 « Programmation dynamique ») pour trouver la solution optimale.

Exercice 7.3 : Allocation de salle de spectacles

On cherche une solution au problème d'allocation d'une salle de spectacles. On définit une liste L contenant pour chaque spectacle d'indice $i \in [\![0, n-1]\!]$ le couple d'entiers (d_i, f_i) où d_i désigne l'heure de début et f_i l'heure de fin : $L = [[d_0, f_0], [d_1, f_1], ..., [d_{n-1}, f_{n-1}]]$. On suppose que les n-uplets de couples (d_i, f_i) sont triés par date de fin f_i croissante. On définit `début` l'heure de début du spectacle et `fin` l'heure de fin du spectacle.

1. On cherche à maximiser le nombre de spectacles dans la salle et non le temps d'occupation. On utilise la méthode intuitive consistant à choisir au fur et à mesure des spectacles dont l'intervalle est compatible avec celui du spectacle précédent et dont l'heure de fin est la plus petite. On suppose que la liste `L` est triée par ordre croissant des heures de fin.

Écrire une fonction itérative `gestion` qui admet comme arguments une liste `L`, un entier `début` (heure de début des spectacles) et un entier `fin` (heure de fin des spectacles). La fonction retourne le nombre maximum de spectacles que l'on peut organiser dans la salle ainsi que la liste des spectacles retenus.

2. Pourquoi cette méthode est-elle appelée gloutonne ?

3. On considère la liste `L1=[[0, 2], [1, 3], [2, 4], [1, 5], [3, 6], [4, 7], [5, 9], [6, 11], [9, 12]]`. Écrire le programme principal permettant d'afficher le nombre maximum de spectacles et la liste des spectacles que l'on peut organiser dans cette salle entre `début` = 1 et `fin` = 12.

Analyse du problème

On utilise la méthode gloutonne, qui consiste à effectuer une succession de choix, chacun d'eux semblant être le meilleur sur le moment.

1.

```
def gestion(L, début, fin):
    # retourne le nombre maximum de spectacles que l'on peut
    # organiser entre début(int) = heure de début et
    # fin(int) = heure de fin dans la salle ainsi que la liste
    # des spectacles retenus L = liste de listes
    # L[i]=[di, fi] = heure de début et de fin du spectacle i
    n=len(L)       # nombre de spectacles
    p=0            # initialisation du nb de spectacles organisés
    LISTE_CONF=[]  # initialisation de la liste des
                   # spectacles organisés
    for i in range(n): # i varie entre 0 inclus et n exclu
        if LISTE_CONF==[] and L[i][0]>=début and L[i][1]<=fin:
            p=p+1
            LISTE_CONF.append(i)    # indice du premier spectacle
                                    # ajouté
        elif p>=1:
            indice=LISTE_CONF[p-1] # indice du dernier spectacle
                                    # ajouté
            if L[i][0]>=L[indice][1] and L[i][1]<=fin:
                p=p+1
                LISTE_CONF.append(i) # indice du spectacle ajouté
    return(p, LISTE_CONF)
```

2. On appelle la méthode gloutonne puisque, à chaque étape, on sélectionne la possibilité qui semble être la meilleure sur le moment, c'est-à-dire qu'on choisit un spectacle qui finit le plus tôt.

On peut montrer que la méthode gloutonne donne ici la solution optimale, mais ce n'est pas toujours le cas, comme on peut le voir dans les exercices sur le rendu de monnaie et sur le sac à dos (exercices 7.1 et 7.2).

3.

```
L1=[[0, 2], [1, 3], [2, 4], [1, 5], [3, 6], [4, 7], [5, 9],\
  [6, 11], [9, 12]]
début=1     # heure de début des spectacles
fin=11      # heure de fin des spectacles
p1, LISTE_CONF1=gestion(L1, début, fin)
print('Nombre de spectacles :', p1)
print("Spectacles que l'on peut organiser :", LISTE_CONF1)
```

Le programme Python affiche :

```
Nombre de spectacles : 3
Spectacles que l'on peut organiser : [1, 4, 7]
```

Partie 6

Lecture et écriture de fichiers – Matrices de pixels et images

Plan

Lecture et écriture de fichiers

8

Exercice 8.1 : Lecture et écriture de fichiers, calculs statistiques

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.readline()` : lecture d'une ligne de l'objet fichier `f`

`\n` : caractère d'échappement : saut de ligne

`c1.strip()` : renvoie une chaîne sans les espaces et les caractères d'échappement (saut de ligne par exemple) en début et fin de la chaîne de caractères `c1`

`c1.split(';')` : sépare une chaîne de caractères (`c1`) en une liste de mots avec le séparateur `';'`

`f.write('exemple')` : écrit dans l'objet fichier `f` la chaîne de caractères `'exemple'`

`f.close()` : ferme le fichier

1. Écrire un programme Python permettant de créer le fichier `'droite.txt'` contenant les éléments suivants :

```
10 : nombre de points du fichier
25;10.9 : abscisse et ordonnée du premier point
20;9.3
15;8.2
12;7.5
9;6.2
6;5.8
3;4.2
```

`0;3.9`

`-3;2.8`

`-6;2` : abscisse et ordonnée du dixième point

2. Écrire un programme Python permettant d'ouvrir un fichier .txt (par exemple `'droite.txt'`) et de récupérer les abscisses et les ordonnées dans deux listes.

3. On cherche à modéliser les n points expérimentaux (x_1, y_1), $(x_2, y_2)\ldots$ (x_n, y_n) par une fonction polynôme du premier ordre, de la forme : $y = ax + b$.

$a = \dfrac{n\sum_{i=1}^{n} x_i y_i - \left(\sum_{i=1}^{n} x_i\right)\left(\sum_{i=1}^{n} y_i\right)}{n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2}$ et $b = \overline{y} - a\overline{x}$ en notant $\overline{x}$ la moyenne des x_i et $\overline{y}$ la moyenne des y_i.

Écrire un programme Python permettant de calculer les coefficients a et b correspondant aux points expérimentaux de la question 2.

4. Écrire un programme Python permettant de représenter graphiquement la fonction $y_{\text{modélisé}} = ax + b$.

Analyse du problème

Cet exercice utilise les fonctions d'écriture et de lecture de fichiers avec Python. Il faut convertir les entiers en chaînes de caractères avant d'utiliser `f.write`. Lors de la lecture du fichier, on parcourt les différentes lignes du fichier avec `f.readline()`. On enlève ensuite les caractères d'échappement (saut de ligne par exemple) et on sépare la chaîne de caractères obtenue en une liste de mots.

1.

```
x=[25,20,15,12,9,6,3,0,-3,-6]
y=[10.9,9.3,8.2,7.5,6.2,5.8,4.2,3.9,2.8,2]
n=len(x)

f=open('droite.txt', 'w')   # création du fichier 'droite.txt'
                            # en écriture
f.write(str(n)+'\n') # première ligne avec la longueur de la liste
for i in range(n):   # i varie entre 0 inclus et n exclu
    f.write(str(x[i])+';'+str(y[i])+'\n')
f.close()            # fermeture du fichier
```

On convertit les entiers et les flottants en chaînes de caractères avant de les insérer dans le fichier. Il ne faut pas oublier le saut de ligne à la fin de chaque ligne du fichier.

2.

```
f=open('droite.txt', 'r')
n=int(f.readline())   # nombre de points
x=[]
y=[]
for i in range(n):    # i varie entre 0 inclus et n exclu
    ligne=f.readline()
    ligne2=ligne.strip()   # enlève les caractères d'échappement
    xchaine,ychaine=ligne2.split(';')
    x.append(float(xchaine))   # conversion de l'abscisse en
                               # type float
            # xchaine est une chaîne de caractères qu'il faut
            # convertir en type float
    y.append(float(ychaine))   # conversion de l'ordonnée en
                               # type float
f.close()   # fermeture du fichier
```

3.

```
sumx=0
sumy=0
sumxy=0
moy_x=0
moy_y=0
sumx2=0
for i in range(n):
    sumx=sumx+x[i]
    sumy=sumy+y[i]
    sumxy=sumxy+x[i]*y[i]
    moy_x+=x[i]
    moy_y+=y[i]
    sumx2+=x[i]**2
moy_x=moy_x/n
moy_y=moy_y/n

a=(n*sumxy-sumx*sumy)/(n*sumx2-sumx**2)
b=moy_y-a*moy_x
```

Le programme Python fournit $a = 0{,}29$ et $b = 3{,}75$.

4.

```
ymodelise=[a*elt+b for elt in x]
plt.plot(x,ymodelise)
```

La première variable dans `plt.plot` est la liste des abscisses des points. La deuxième variable est la liste des ordonnées des points.

Exercice 8.2 : Lecture de fichiers

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()` : transforme toutes les lignes de l'objet fichier `f` en une liste de chaînes de caractères

`c1.strip()` : renvoie une chaîne sans les espaces et les caractères d'échappement (saut de ligne par exemple) en début et fin de la chaîne de caractères `c1`

`c1.split(';')` : sépare une chaîne de caractères (`c1`) en une liste de mots avec le séparateur `';'`

`f.write('exemple')` : écrit dans l'objet fichier `f` la chaîne de caractères `'exemple'`

`f.close()` : ferme le fichier

On cherche à calculer une valeur approchée de l'intégrale d'une fonction donnée par des points dont les coordonnées sont situées dans un fichier.

1. Le fichier «`ex_001.txt`» contient une quinzaine de lignes selon le modèle suivant :

```
0.0;0.0
0.111111111111;0.0122949573053
0.222222222222;0.0485751653206
```

Chaque ligne contient deux valeurs flottantes séparées par un point-virgule, représentant respectivement l'abscisse et l'ordonnée d'un point. Les points sont ordonnés dans l'ordre croissant de leurs abscisses.

Écrire un programme Python permettant d'ouvrir le fichier en lecture, de le lire et de construire la liste `X` des abscisses et la liste `Y` des ordonnées contenues dans ce fichier.

2. Écrire un programme Python permettant de représenter les points sur une figure.

3. Les points précédents sont situés sur la courbe représentative d'une fonction *f*. On souhaite déterminer une valeur approchée de l'intégrale *I* de cette fonction sur le segment où elle est définie.

Écrire une fonction `trapeze`, d'arguments deux listes `Y` et `X` de même longueur *n*, renvoyant :

$$\sum_{i=1}^{n-1}(X_i - X_{i-1})\frac{Y_i + Y_{i-1}}{2}$$

`trapeze(Y, X)` renvoie donc une valeur approchée de l'intégrale *I* par la méthode des trapèzes.

Analyse du problème

Cet exercice utilise les fonctions de lecture de fichiers avec Python. Dans l'exercice précédent, on lit les lignes du fichier au fur et à mesure en utilisant la méthode `f.readline()`. Ici on récupère directement une liste de chaînes de caractères avec la méthode `f.readlines()`. Il faut alors parcourir cette liste pour récupérer les différentes lignes du fichier. Pour chaque ligne, on enlève les caractères d'échappement (saut de ligne par exemple) et on sépare la chaîne de caractères obtenue en une liste de mots.

1. On récupère dans la variable `data` une liste de chaînes de caractères. Pour parcourir cette liste, on utilise l'instruction `for chaine in data`.

```
f=open("ex_001.txt", 'r')   # ouverture du fichier en lecture
data=f.readlines()   # data contient une liste de chaînes
                     # de caractères
    # chaque chaîne de caractères correspond à une ligne
    # du fichier
X, Y=[], []   # création des listes vides X et Y
for chaine in data:   # on parcourt la liste de chaînes de
                      # caractères
    chaine2=chaine.strip()   # enlève les caractères d'échappement
    abs,ord=chaine2.split(';')
    X.append(float(abs))   # conversion de l'abscisse en type
                           # float
            # on ajoute cette abscisse dans la liste X
    Y.append(float(ord))   # conversion de l'ordonnée en type
                           # float
            # on ajoute cette ordonnée dans la liste Y
f.close()   # fermeture du fichier
```

Cours :

Pour initialiser une liste, on utilise `X=[]`.

Pour ajouter des éléments dans une liste, on utilise `X.append(valeur)`.

Pour supprimer le dernier élément d'une liste, on utilise `X.pop()`.

Il faut bien connaître les arguments de la fonction `plt.plot(X, Y)` : la première liste `X` représente l'abscisse des points, la deuxième liste `Y` représente l'ordonnée des points.

2.

```
plt.figure()   # nouvelle fenêtre graphique
plt.plot(X, Y)
    # ou utilisation de la fonction ci-dessous, qui permet
    # de mettre  une croix pour les points expérimentaux
    # et de relier les points entre eux :
    # plt.plot(LX, LY, '+', linestyle="-")
plt.show()     # affiche la figure à l'écran
```

Cours :
Il faut bien connaître la syntaxe de `range` :

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

`i` varie entre `start` inclus et `stop` exclu avec un pas égal à `step`.

Lorsque `step` n'est pas indiqué, le pas vaut 1 par défaut.

3.

```
def trapeze(Y, X):
    # la fonction renvoie l'intégrale I par la méthode des
    # trapèzes
    n=len(X)   # nombre d'éléments de la liste
    I=0   # initialisation de la variable I
    for i in range(1, n):   # i varie entre 1 inclus et n exclu
        I+=(X[i]-X[i-1])*(Y[i]+Y[i-1])/2
    return I

print(trapeze(Y, X))
```

On pourrait écrire :

```
I=I+(X[i]-X[i-1])*(Y[i]+Y[i-1])/2
```

au lieu de :

```
I+=(X[i]-X[i-1])*(Y[i]+Y[i-1])/2
```

9 Matrices de pixels et images, traitement d'images

Exercice 9.1 : Traitement d'images et filtrage passe-bas

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())
```

1. Écrire une fonction `creation_mat` qui admet comme argument l'image en niveaux de gris `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…

2. Écrire une fonction `inv_contraste` qui admet comme argument `M` la matrice représentant une image. Cette fonction retourne une nouvelle matrice représentant la moitié inférieure de l'image avec une inversion du contraste.

3. Écrire une fonction `trois_niveaux_gris` qui admet comme argument `M` la matrice représentant une image. Cette fonction retourne une nouvelle matrice avec 3 niveaux de gris uniquement : les niveaux de gris entre 0 et 80 inclus sont remplacés par 60, les niveaux de gris entre 80 et 150 inclus sont remplacés par 120 et les autres niveaux de gris sont remplacés par 220.

4. Écrire une fonction `reduction` qui admet comme argument `M` la matrice représentant une image. Cette fonction retourne une nouvelle matrice en ne gardant qu'un pixel sur 3 de l'image pour la largeur et la hauteur.

5. On souhaite réaliser un filtre passe-bas qui adoucit les détails d'une image représentée par la matrice `M`. On considère la matrice `A` :

1/12	1/12	1/12
1/12	4/12	1/12
1/12	1/12	1/12

Le traitement suivant est appliqué à 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 `filtrage` qui admet comme argument `M` la matrice représentant une image. Cette fonction retourne une nouvelle matrice résultat du filtrage passe-bas de l'image.

Analyse du problème

On repère un pixel par (*i, j*) où *i* désigne l'indice de la ligne et *j* l'indice de la colonne. Pour chaque pixel, on a un niveau de gris compris entre 0 et 255. On obtient ainsi une matrice dont chaque valeur correspond au niveau de gris. On peut alors modifier facilement les valeurs de la matrice pour effectuer un traitement d'images.

Cours :
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].

Utilisation d'une liste de listes

Matrice

On représente une matrice 2×3 par la 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 `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
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 la 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 800×600 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 la 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
```

1.

```
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
```

2.

```
def inv_contraste(M):
    # argument d'entrée : matrice M (liste de listes)
    # en niveaux de gris
    # la fonction retourne M2 la moitié inférieure de l'image M
    # avec inversion du contraste
    n=len(M)                  # n = nombre de lignes   (hauteur)
    n2=n//2
    p=len(M[0])               # p = nombre de colonnes (largeur)
    M2=[[0 for j in range(p)] for i in range(n2, n)]
    for i in range(n2, n):
        for j in range(p):
            M2[i-n2][j]=255-M[i][j]  # inversion du contraste
    return M2
```

3.

```
def trois_niveaux_gris(M):
    # la fonction renvoie une matrice M2 (liste de listes)
    # avec 3 niveaux de gris uniquement
    # 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(n):
        for j in range(p):
            if M[i][j]<=80:      # teste si le niveau de gris
                                 # est <= 80
                M2[i][j]=60
            elif M[i][j]<=150 :  # teste si le niveau de gris
                                 # est <= 150
```

```
                M2[i][j]=120
            else:                # le niveau de gris est > 150
                M2[i][j]=220
    return M2
```

On pourrait écrire `elif M[i][j]>80 and M[i][j]<=150`. C'est inutile car si la première condition est vérifiée, Python ne teste pas l'instruction après `elif`.

4.

```
def reduction(M):
    # la fonction renvoie une matrice M2 (liste de listes)
    # avec 1 pixel sur 3 pour la largeur et la hauteur
    # 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(0, p, 3)] for i in range(0, n, 3)]
    for i in range(0, n, 3):
        for j in range(0, p, 3):
            M2[i//3][j//3]=M[i][j]
    return M2
```

5.

```
def somtab(A):                # somme de tous les éléments de
                              # la matrice A (liste de listes)
    n=len(A)                  # n = nombre de lignes   (hauteur)
    p=len(A[0])               # p = nombre de colonnes (largeur)
    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   (hauteur)
    p=len(A[0])               # p = nombre de colonnes (largeur)
    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 filtrage(M):
    # la fonction renvoie une matrice M2 (liste de listes)
    # filtrage passe-bas de la matrice M
    # 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)]
```

```
A=[[1/12,1/12,1/12],[1/12,4/12,1/12],[1/12,1/12,1/12]]
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
```

Remarque : Voir le site Dunod pour télécharger les programmes Python avec des exemples d'images.

Exercice 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())
```

1. On souhaite réaliser un lissage de l'image représentée par la matrice `M` en niveaux de gris. 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` : $\mathbf{A}_{0,0} \times \mathbf{M}_{i-1,j-1} + \mathbf{A}_{1,0} \times \mathbf{M}_{i,j-1} + \mathbf{A}_{2,0} \times \mathbf{M}_{i+1,j-1} + \ldots$

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 3×3. Cette fonction retourne une nouvelle matrice résultat du filtrage de l'image.

2. 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 `M2` la matrice de l'image ainsi filtrée.

3. Proposer un filtre permettant de détecter le contour selon la direction verticale. On appelle `M3` la matrice de l'image ainsi filtrée.

4. Proposer un filtre permettant de détecter les contours dans les deux directions en utilisant pour chaque point de l'image $\sqrt{\left(\mathbf{M2}_{i,j}\right)^2 + \left(\mathbf{M3}_{i,j}\right)^2}$.

Analyse du problème

On repère un pixel par (i, j) où i désigne l'indice de la ligne et j l'indice de la colonne. Pour chaque pixel, on a un niveau de gris compris entre 0 et 255. On utilise deux boucles `for` pour décrire tous les pixels de l'image.

1.

```
def somtab(A):                  # somme de tous les éléments de
                                # la matrice A (liste de listes)
    n=len(A)                    # n = nombre de lignes   (hauteur)
    p=len(A[0])                 # p = nombre de colonnes (largeur)
    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   (hauteur)
    p=len(A[0])                 # p = nombre de colonnes (largeur)
    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
```

2. 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]]
M2=filtre(M, A)
```

3. La dérivée dans la direction verticale 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 verticale :

0	−1	0
0	1	0
0	0	0

```
A=[[0,-1,0],[0,1,0],[0,0,0]]
M3=filtre(M, A)
```

4.

```
def contour(M2, M3):
    import math as 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+(M3[i][j])**2)
    return M5
```

Remarque : Voir le site Dunod pour télécharger les programmes Python avec des exemples d'images.

Partie 7

Tris

Plan

Tris

10

Exercice 10.1 : Tri par insertion

Le tri par insertion est souvent utilisé pour trier des cartes. Il consiste à insérer les éléments d'une partie de la liste non triée dans la liste triée.

Présentation du tri par insertion :

On considère la liste non triée : L=[8][5][3][9][2] comprenant $n = 5$ éléments. Avec Python, on a : $L[0] = 8$ … et $L[n-1] = 2$. On parcourt la liste du deuxième au dernier élément. Lorsqu'on est à l'étape k (k variant de 1 à n–1, les éléments précédents $L[k]$ sont déjà triés. Il faut donc insérer cet élément d'indice k dans la liste triée.

Mise en place de l'algorithme :

On envisage deux boucles pour réaliser le tri par insertion :

- **Première boucle d'indice *k*** (k variant de 1 à n–1). Quand on considère l'élément d'indice k, on considère que les éléments précédents sont déjà triés.

  Par exemple, pour $k = 2$. On a la liste suivante : [5][8][3][9][2] avec L[2]=3.

  Les éléments avant L[k] sont déjà triés : [5][8].

  On pose x = L[k] = 3.

- **Deuxième boucle d'indice *i*** (i variant de k–1 à 0). Il faut trouver où l'élément x doit être inséré dans cette liste triée ([5][8] dans l'exemple). On compare $L[i]$ à x. Si $L[i] > x$, alors il faut décaler cet élément vers la droite : $L[i+1] = L[i]$. Sinon, il suffit de mettre x dans la valeur du trou qui a été laissé.

Exemple :

Différentes étapes pour la boucle d'indice k :

- liste non triée : [8][5][3][9][2]
- $k = 1$: [8][5][3][9][2] → [5][8][3][9][2]
- $k = 2$: [5][8][3][9][2] → [3][5][8][9][2]
- $k = 3$: [3][5][8][9][2] → [3][5][8][9][2]

• $k = 4$: 3 5 8 9 2 → 2 3 5 8 9
(k=4 sous le 2 ; k=4 sous le 9)

On considère une liste non triée d'entiers ou de flottants.

1. Écrire une fonction `tri_insertion` qui admet comme argument une liste `L` et permet de la trier par ordre croissant en utilisant la méthode du tri par insertion.

2. Écrire le programme principal permettant de trier la liste `L=[8, 5, 3, 9, 2]`.

3. Partant d'une liste de couples (entier, chaîne de caractères), on souhaite trier la liste `L2` par ordre croissant de la population en millions d'habitants : `L2=[[67, 'France'], [40, 'Irak'], [47, 'Kenya'], [32, 'Pérou'], [66, 'Royaume-Uni'], [66, 'Thaïlande']]`. Écrire une fonction `tri_insertion2` permettant de trier la liste `L2` par ordre croissant de la population.

4. Donner les caractéristiques du tri par insertion.

Analyse du problème

On étudie dans cet exercice le tri par insertion qui est un tri en place car il n'utilise pas de liste auxiliaire. Sa complexité spatiale est faible.

1.

```
def tri_insertion(L):
    # la fonction trie par ordre croissant la liste L
    n=len(L)
    for k in range(1, n): # k varie entre 1 inclus et n exclu
        i=k-1   # deuxième boucle démarre à k-1
        # les éléments entre 0 et k-1 sont triés
        x=L[k]  # mémorisation de la valeur de L[k]
        while i>=0 and L[i]>x:
            L[i+1]=L[i] # décale les éléments de la liste
            i=i-1       # décrémente de 1 la valeur de i
        L[i+1]=x        # met la valeur dans le trou
    # return L est inutile car L est passé en référence
```

Si on modifie un argument d'entrée muable (liste, dictionnaire, deque) dans une fonction alors on ne retrouve pas l'état initial de l'objet lorsqu'on quitte la fonction (voir exercice 1.5 « Passage par référence pour les listes, effet de bord » dans le chapitre « Prise en main de Python »).

La variable `L` dans la fonction `tri_insertion` est une liste qui est un objet muable. Il est donc inutile de retourner la variable `L` dans cette fonction !

2.

```
L=[8, 5, 3, 9, 2]
print(L)
tri_insertion(L)
print(L)
```

Python affiche :

```
[2, 3, 5, 8, 9]
```

Remarques :

Le programme principal suivant affiche `None` puisqu'il n'y a pas de `return`.

```
print(tri_insertion([5, 2, 3, 1, 4]))  # affiche None

def tri_insertion_test(L):
    # la fonction trie par ordre croissant la liste L
    n=len(L)
    for k in range(1, n): # k varie entre 1 inclus et n exclu
        i=k-1   # deuxième boucle démarre à k-1
        # les éléments entre 0 et k-1 sont triés
        x=L[k]  # mémorisation de la valeur de L[k]
        while i>=0 and L[i]>x:
            L[i+1]=L[i] # décale les éléments de la liste
            i=i-1       # décrémente de 1 la valeur de i
        L[i+1]=x        # met la valeur dans le trou
    return      # ne retourne pas de variable
```

Le programme principal suivant affiche `None` puisque `return` ne retourne pas de variable.

```
print(tri_insertion_test ([5, 2, 3, 1, 4]))  # affiche None
```

L'instruction `return` est inutile dans cette fonction.

3.

```
def tri_insertion2(L):
    # la fonction trie par ordre croissant la liste L
    # L[i][0] valeur à trier
    # attention : une liste Python est une variable passée
    # par référence
    n=len(L)
    for k in range(1, n): # k varie entre 1 inclus et n exclu
        i=k-1             # deuxième boucle démarre à k-1
                          # les éléments entre 0 et k-1 sont triés
        x, pays=L[k]      # mémorisation de la valeur de L[k]
        while i>=0 and L[i][0]>x:
            L[i+1]=L[i]   # décale les éléments de la liste
            i=i-1         # décrémente de 1 la valeur de i
        L[i+1]=[x, pays]  # met la valeur dans le trou
    # return L est inutile car L est passé en référence
L2=[[67, 'France'], [40, 'Irak'],[47, 'Kenya'],\
  [32, 'Pérou'], [66, 'Royaume-Uni'], [66, 'Thaïlande']]
print(L2)
tri_insertion2(L2)
print(L2)
```

Cours :
On cherche à trier un ensemble d'éléments, c'est-à-dire à les ordonner en fonction d'une relation d'ordre définie sur ces éléments.

- Un tri comparatif est basé sur la comparaison des éléments entre eux.
- Un tri itératif est basé sur un ou plusieurs parcours itératifs de la liste.
- Un tri récursif est basé sur une procédure récursive.
- Un tri en place n'utilise qu'un espace mémoire de taille constante en plus de l'espace servant à stocker les éléments à trier. Il n'utilise pas de liste auxiliaire.
- Un tri stable conserve l'ordre initial des éléments de même clé. Deux éléments avec des clés égales apparaîtront dans le même ordre dans la liste triée et dans la liste non triée.

On rencontre différents algorithmes de tri :

- Tri par insertion : tri comparatif, itératif, stable. Tri en place.
- Tri par sélection : tri comparatif, itératif, instable. 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.

4. Le tri par insertion est comparatif et itératif.

La liste initiale est triée par ordre alphabétique : `[[67, 'France'], [40, 'Irak'], [47, 'Kenya'], [32, 'Pérou'], [66, 'Royaume-Uni'], [66, 'Thaïlande']]`.

La liste triée par ordre croissant de la population est : `[[32, 'Pérou'], [40, 'Irak'], [47, 'Kenya'], [66, 'Royaume-Uni'], [66, 'Thaïlande'], [67, 'France']]`.

Le tri par insertion est stable puisqu'on garde l'ordre alphabétique dans la liste triée pour les pays qui ont la même population.

Le tri par insertion est un tri en place car il n'utilise pas de liste auxiliaire.

Exercice 10.2 : Tri par sélection

On considère une liste `L` contenant n éléments. Le tri par sélection consiste à :

- rechercher le plus petit élément de la liste et le placer en première position ;
- rechercher le deuxième plus petit élément de la liste et le placer en deuxième position ;
- répéter itérativement le processus tel que la liste soit entièrement triée.

Mise en place de l'algorithme :

On parcourt la liste `L[i]` en faisant varier i entre 0 inclus et $n-2$ inclus :

- On cherche `ind_mini` l'indice correspondant à l'élément le plus petit de la liste `L[i : n]`.
- Si `ind_mini` est différent de i, alors on permute `L[i]` avec `L[ind_mini]`.

1. Écrire une fonction `tri_sélection` qui admet comme argument une liste `L` et permet de la trier par ordre croissant en utilisant la méthode du tri par sélection. Écrire le programme principal permettant de trier la liste `L=[8, 5, 3, 9, 2]`.

2. Partant d'une liste de couples (entier, chaîne de caractères), on souhaite trier la liste `L2` par ordre croissant de la population en millions d'habitants : `L2=[[67, 'France'], [40, 'Irak'], [47, 'Kenya'], [32, 'Pérou'], [66, 'Royaume-Uni'], [66, 'Thaïlande']]`. Écrire une fonction `tri_sélection2` permettant de trier la liste `L2` par ordre croissant de la population.

3. Donner les caractéristiques du tri par sélection.

4. Évaluer la complexité dans le pire des cas lors de l'appel de la fonction `tri_sélection`.

Analyse du problème

On étudie dans cet exercice le tri par sélection, qui est un tri en place car il n'utilise pas de liste auxiliaire. Le premier élément d'une liste `L` a pour indice 0 avec Python.

1.

```
def tri_sélection(L):
    # la fonction trie par ordre croissant la liste L
    n=len(L)
    for i in range(0, n-1):
        # recherche du minimum de la liste L[i: n]
        mini=L[i]
        ind_mini=i
        for j in range(i+1, n):
            if L[j]<mini:
                ind_mini=j
                mini=L[j]
        # on permute L[i] et L[ind_mini] si on a trouvé
        # un nouveau minimum
        if ind_mini!=i: # ind_mini est différent de i
            L[i], L[ind_mini]=L[ind_mini], L[i]
        # return L est inutile car L est passé en référence

L=[8, 5, 3, 10, 2, 9]
tri_sélection(L)
print(L)
```

2.

```
def tri_sélection2(L):
    # la fonction trie par ordre croissant la liste L
    # L[i][0] valeur à trier
    n=len(L)
    for i in range(0, n-1):
        # recherche du minimum de la liste L[i: n]
```

```
        mini=L[i][0]
        ind_mini=i
        for j in range(i+1, n): # parcourt L[j] pour j>i
            if L[j][0]<mini:
                ind_mini=j
                mini=L[j][0]
        # on permute L[i] et L[ind_mini] si on a trouvé
        # un nouveau minimum
        if ind_mini!=i:         # ind_mini est différent de i
            L[i], L[ind_mini]=L[ind_mini], L[i]
        # return L est inutile car L est passé en référence
```

3. Le tri par sélection est comparatif, instable et itératif. C'est un tri en place car il n'utilise pas de liste auxiliaire.

4. On cherche à calculer le nombre d'opérations élémentaires :

- Ligne `n = len(L)` : 2 opérations élémentaires (appel de `len(L)` et affectation).
- Boucle `for i in range(n-1)` :
  - 3 opérations élémentaires (appel de l'élément `L[i]`, affectation de `mini`, affectation de `ind_mini`) ;
  - boucle `for j in range(i+1, n)` : on se place dans le pire des cas, on a 5 opérations élémentaires (appel de l'élément `L[j]`, test, affectation, appel de l'élément `L[j]` et affectation) ;
  - dans le pire des cas, on a 6 opérations élémentaires (test, appel de l'élément `L[i]`, appel de l'élément `L[ind_mini]` et 3 affectations).

Le nombre total d'opérations élémentaires vaut :

$$2+\sum_{i=0}^{n-2}\left(3+5\left((n-1)-(i+1)\right)+6\right)=-2+\frac{3n}{2}+\frac{5}{2}n^2$$

La complexité est quadratique en $O\left(n^2\right)$.

Remarque :

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

On peut montrer que la complexité est quadratique dans tous les cas.

Exercice 10.3 : Tri rapide (sauf TSI et TPC)

On considère la liste `L` = 10 3 9 6 8 non triée. On modifie la liste `L` en place. Lors des différents appels récursifs, on travaille sur une partie de la liste `L`. On repère les éléments d'une sous-liste par les pointeurs *a* et *b*.

Fonction pivot :

- On choisit le dernier élément de la liste qui est appelé le pivot *p* (ici *p* = 8) : 10 3 9 6 8 (p)
- On réordonne la liste en plaçant tous les éléments inférieurs ou égaux au pivot à gauche de celui-ci et les éléments strictement supérieurs à droite du pivot. Le pivot *p* est alors à sa bonne place dans la liste à trier.

  L'indice du pivot `ind_p` vaut 0 au début de la procédure.
- On compare `L[0]` = 10 au pivot *p* = 8. Comme `L[0]` > *p*, pas de modification : `ind_p` = 0. 10 3 9 6 8
- On compare `L[1]` = 3 au pivot *p* = 8. Comme `L[1]` <= *p*, on échange `L[1]` et `L[ind_p]` et on incrémente `ind_p`. On a alors : `ind_p` = 1. 3 10 9 6 8
- On compare `L[2]` = 9 au pivot *p* = 8. Comme `L[2]` > *p*, pas de changement : `ind_p` = 1. 3 10 9 6 8
- On compare `L[3]` = 6 au pivot *p* = 8. Comme `L[3]` <= *p*, on échange `L[3]` et `L[ind_p]` et on incrémente `ind_p`. On a alors : `ind_p` = 2. 3 6 9 10 8
- Pour cette dernière étape, on ne modifie pas `ind_p` mais on permute le dernier élément et `L[ind_p]`. 3 6 8 10 9

  On est sûr que le pivot est à la bonne place.

Algorithme principal :

Il reste à appliquer la procédure précédente aux deux sous-listes à gauche et à droite du pivot, c'est-à-dire 3 6 et 10 9.

On a une procédure récursive puisqu'on applique la fonction `pivot` à deux sous-listes. On a des sous-listes de longueur de plus en plus petite. On arrive à une sous-liste comportant un seul élément, qui est donc triée (condition d'arrêt de la fonction récursive) !

1. Écrire une fonction `pivot` qui admet comme arguments une liste `L` et deux indices `a` et `b` permettant de définir une sous-liste de `L` avec des indices compris entre `a` et `b`. Le dernier élément de la sous-liste est appelé le pivot. Cette fonction réordonne les éléments de la sous-liste en plaçant tous les éléments inférieurs ou égaux au pivot à gauche de celui-ci et les éléments strictement supérieurs au pivot à droite de celui-ci. Cette fonction retourne la position du pivot dans la liste.

2. Écrire une fonction récursive `tri_rapide` permettant de trier une liste par ordre croissant en utilisant la fonction `pivot`.

3. Écrire le programme principal permettant de trier la liste `L=[10, 3, 9, 6, 8]`. Représenter l'arbre des appels de la fonction récursive `tri_rapide`.

4. Donner les caractéristiques du tri rapide.

Analyse du problème

Le tri rapide s'appuie sur le principe « diviser pour régner » comme le tri par partition-fusion. On réalise un tri en place.

1.

```
def pivot(L, a, b): # a = indice de début, et b = indice de fin
    # la fonction renvoie la position du pivot dans la liste
    # éléments inférieurs ou égaux sont à gauche du pivot
    # L[i][0] valeur à trier
    p=L[b] # valeur du pivot = dernier élément de la liste
    ind_p=a                    # indice du pivot
    for i in range(a, b):      # i varie entre a inclus et b exclu
        if L[i]<=p:
            L[i], L[ind_p]=L[ind_p], L[i] # on échange les
                                          # 2 éléments
            ind_p+=1
    L[b], L[ind_p]=L[ind_p], L[b] # échange les 2 éléments
        # inutile de retourner L car passage par référence
        # la valeur L[ind_p] est bien placée dans la liste à trier
    return ind_p
```

Si on modifie un argument d'entrée muable (liste, dictionnaire, deque) dans une fonction alors on ne retrouve pas l'état initial de l'objet lorsqu'on quitte la fonction (voir exercice 1.5 « Passage par référence pour les listes, effet de bord » dans le chapitre « Prise en main de Python »).

La variable `L` dans la fonction `pivot` est une liste qui est un objet muable. Il est donc inutile de retourner la variable `L` dans cette fonction !

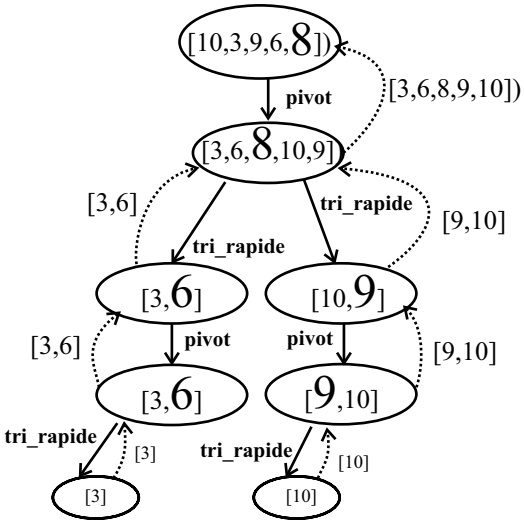
2.

```
def tri_rapide(L, a, b):
    # a = indice de début, et b = indice de fin
    # la fonction trie par ordre croissant la liste L
    if b>a:
        ind_p=pivot(L, a, b)
        tri_rapide(L, a, ind_p-1) # tri_rapide pour les indices
                                  # entre a et ind_p-1
        tri_rapide(L, ind_p+1, b) # tri rapide pour les indices
                                  # entre ind_p+1 et b
    # si a=b alors la sous-liste est triée : condition d'arrêt
    # si a>b pas de changement. Il faut bien considérer ce cas
    # comme condition d'arrêt
```

3.

```
L=[10, 3, 9, 6, 8]
n=len(L)
print(L)
tri_rapide(L, 0, n-1)  # la fonction retourne none
                       # puisque L est triée en place
print(L)
```

L'arbre ci-dessous représente les différents appels de la fonction `tri_rapide`. La valeur du pivot est représentée avec une taille de police de caractères plus grande.



Remarque :

Lorsqu'on applique la fonction `tri_rapide` à la sous-liste [10, 9] avec $a = 3$ et $b = 4$, il y a trois actions :

- appel de la fonction `pivot` : le pivot vaut 9 et l'indice du pivot vaut `ind_p` = 3 puisque 9 fait partie de la liste `L` = [3, 6, 8, 9, 10] ;
- appel de la fonction `tri_rapide` à la sous-liste définie par $a = 3$ et `ind_p` – 1 = 2. La fonction `tri_rapide` ne modifie rien : c'est une condition d'arrêt ;
- appel de la fonction `tri_rapide` à la sous-liste définie par `ind_p` + 1 = 4 et $b = 4$. La fonction `tri_rapide` ne modifie rien puisque la sous-liste constituée d'un seul élément est déjà triée : c'est une condition d'arrêt.

La complexité du tri rapide est quasi linéaire en $O(n \log n)$. Le tri rapide est plus efficace que le tri par insertion dont la complexité est quadratique en $O(n^2)$ dans le pire des cas.

4. Les caractéristiques du tri rapide sont : comparatif, récursif et instable. Le tri rapide est en place.

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

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

Exercice 10.4 : Tri par partition-fusion

On considère une liste `L` non triée d'entiers ou de flottants. On cherche à trier cette liste par ordre croissant en utilisant la méthode du tri par partition-fusion.

Algorithme principal :

On partage la liste initiale `L` de longueur n en deux sous-listes `L1` et `L2` de longueurs $n//2$ et $n-n//2$. On trie de façon récursive les deux sous-listes puis on fusionne les deux sous-listes triées.

Fusion de deux sous-listes L1 et L2 triées :

On considère par exemple `L1=[3, 5, 8]` et `L2=[4, 6]`.

On construit la nouvelle liste `L` en retirant le premier élément de la première liste ou de la deuxième liste.

- Première étape : on considère le premier élément de chaque liste : 3 5 8 et 4 6.

  On cherche le minimum de 3 et 4 que l'on ajoute dans la nouvelle liste : `L=[3]`.

  Il reste alors `L1=[5, 8]` et `L2=[4, 6]`.

- Deuxième étape : on considère le premier élément de chaque liste : 5 8 et 4 6.

  On cherche le minimum de 5 et 4 que l'on ajoute dans la nouvelle liste : `L=[3, 4]`.

  Il reste alors `L1=[5, 8]` et `L2=[6]`.

- Étapes suivantes :

  `L=[3, 4, 5]` ; `L1=[8]` et `L2=[6]`

  `L=[3, 4, 5, 6]` ; `L1=[8]` et `L2=[]`

  `L=[3, 4, 5, 6, 8]` ; `L1=[]` et `L2=[]`

On obtient la liste `L` triée.

1. Écrire une fonction itérative `fusion` qui admet comme arguments deux listes `L1` et `L2` triées et retourne la fusion triée des deux listes.

2. Réécrire une version récursive de la fonction précédente que l'on appellera `fusion_rec`.

3. Écrire une fonction récursive `tri_fusion` permettant de trier la liste `L`. On pourra partager la liste initiale `L` en deux sous-listes `L1` et `L2`.

4. Écrire le programme principal permettant de trier par ordre croissant la liste `L=[8, 3, 5, 1, 9, 5, 12, 15]`. La fonction `tri_fusion` comporte plusieurs appels récursifs. Représenter l'arbre des appels de la fonction `tri_fusion` pour la liste `L`.

5. Donner les caractéristiques du tri par partition-fusion.

Analyse du problème

Le tri par partition-fusion s'appuie sur le principe « diviser pour régner », c'est-à-dire que l'on divise (partitionne) le problème en deux sous-problèmes que l'on sait résoudre. Il reste à utiliser les deux solutions pour résoudre le problème initial.

1.

```
def fusion(L1, L2):
    # la fonction L retourne la fusion triée des deux listes
    # L1 et L2
    i1, i2=0, 0                 # position du pointeur de chaque
                                # liste
    n1, n2=len(L1), len(L2)     # longueur des listes
    n=n1+n2                     # longueur de L1+L2
    L=[]
    while i1+i2<n:              # il faut parcourir tous les
                                # éléments de L1+L2
        if i1==n1:              # la liste L1 est parcourue
                                # entièrement
            return L+L2[i2:]    # il faut ajouter les éléments
                                # restants de L2
        elif i2==n2:            # la liste L2 est parcourue
                                # entièrement
            return L+L1[i1:]    # il faut ajouter les éléments
                                # restants de L1
        elif L1[i1]<L2[i2]:
            L=L+[L1[i1]]        # ajoute L1[i1]
            i1+=1               # incrémente de 1 le pointeur de L1
        else:                   # on a forcément L1[i1]>=L2[i2]
            L=L+[L2[i2]]
            i2+=1
    return L
```

Il faut bien connaître le slicing ou extraction de tranche pour les listes : instruction `L[start:stop]` (voir exercice 1.6 « Slicing, extraction de tranche, » dans le chapitre « Prise en main de Python »).

- `start` désigne l'indice de départ.
- `stop–start` désigne la longueur de la liste extraite (lorsque le pas vaut 1). L'indice final vaut `stop–1` !

Remarque : Il est préférable d'utiliser deux pointeurs `i1` et `i2` plutôt que d'enlever au fur et à mesure les éléments de `L1` et `L2` quand on construit la nouvelle liste `L`. On ne modifie pas les listes `L1` et `L2` en utilisant les pointeurs `i1` et `i2`.

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

2.

```
def fusion_rec(L1, L2):
    # la fonction retourne la fusion triée des deux listes
    # L1 et L2
    if L1==[]:
        return L2   # condition d'arrêt
    elif L2==[]:
        return L1   # condition d'arrêt
    elif L1[0]<L2[0]:
        return ([L1[0]]+fusion_rec(L1[1:], L2))
    else:
        return ([L2[0]]+fusion_rec(L1, L2[1:]))
```

3.

```
def tri_fusion(L):
    # la fonction trie par ordre croissant la liste L
    n=len(L)
    if n==1:
        return L    # condition d'arrêt
    else:
        L1=L[:n//2] # indices compris entre 0 inclus et n//2 exclu
                    # la longueur de L1 vaut n//2
        L2=L[n//2:] # indices compris entre n//2 inclus et n exclu
        return fusion_rec(tri_fusion(L1), tri_fusion(L2))
```

4.

```
L=[8, 3, 5, 1, 9, 5, 12, 15]
L_tri=tri_fusion(L)
print(L_tri)
```

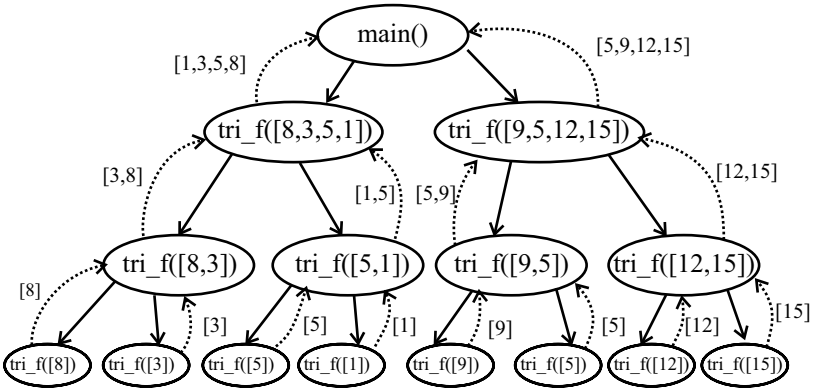
Remarque :

À chaque appel de la fonction récursive, on coupe la liste en deux. On arrive toujours à une sous-liste comportant un seul élément, qui est donc triée (condition d'arrêt de la fonction récursive) !

La complexité spatiale est très mauvaise puisqu'on utilise une fonction récursive qui appelle elle-même une fonction récursive. Le tri n'est pas en place comme avec le tri par insertion.

1er appel de la fonction `tri_fusion` : `L1=[8, 3, 5, 1]` et `L2=[9, 5, 12, 15]`. Avant de fusionner `L1` et `L2`, il faut les trier de façon récursive en appelant la fonction `tri_fusion`.

L'arbre ci-dessous représente les différents appels de la fonction `tri_fusion` notée `tri_f`.



Remarque : La complexité du tri par partition-fusion est quasi linéaire en $O(n \log n)$. Le tri par partition-fusion est plus efficace que le tri par insertion dont la complexité est quadratique en $O(n^2)$dans le pire des cas.

5. Les caractéristiques du tri par partition-fusion sont : comparatif, récursif, stable. Le tri n'est pas en place.

On utilise la méthode « diviser pour régner », qui peut se décomposer en trois étapes :

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

Exercice 10.5 : Tri par comptage, histogramme

Le tri par comptage est un algorithme de tri d'entiers. On considère des entiers de 0 à *p* dans une liste `L` contenant *n* éléments : `L=[10, 20, 12, 12, 16, 16, 12, 20, 15]`. L'algorithme compte le nombre d'occurrences de chaque entier.

Les fonctions suivantes permettent le tracé d'histogrammes :

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

1. Écrire une fonction `tri_comptage` réalisant cette opération.

2. Donner les caractéristiques du tri par comptage.

3. Évaluer la complexité dans le pire des cas lors de l'appel de la fonction `tri_comptage` en fonction de *n* et *p*.

4. Afficher graphiquement l'histogramme de la liste `L`. L'histogramme doit avoir les caractéristiques suivantes :

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

5. Proposer une amélioration de la fonction `tri_comptage` en tenant compte du minimum et du maximum de la liste `L`.

Analyse du problème

On étudie dans cet exercice le tri par comptage. On crée une liste `HISTO` qui représente l'histogramme (ou liste de comptage) des éléments de `L`. Voir exercice 2.2 « Tracé d'un histogramme avec matplotlib » dans le chapitre « Graphiques ».

1.

```
def tri_comptage(L):
    # la fonction retourne L_tri, qui est la liste L triée par
    # ordre croissant
```

```
    n=len(L)              # nombre d'éléments de L
    L_tri=[]
    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].

2. Le tri par comptage n'est pas comparatif (on ne compare pas les éléments de la liste entre eux). Le tri est itératif. Le tri n'est pas en place puisqu'il utilise 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.

3. On cherche à calculer le nombre d'opérations élémentaires :

- Ligne `n=len(L)` : 2 opérations élémentaires (appel de `len(L)` et affectation).
- Ligne `L_tri=[]` : 1 opération élémentaire.
- Boucle `for i in range(n)` : à chaque étape, 4 opérations élémentaires (appel de l'élément `L[i]`, test, appel de l'élément `L[i]` et affectation.
  On a donc 4*n* opérations élémentaires.
- Ligne `HISTO=[0 for i in range(p+1)]` : *p*+1 opérations élémentaires.

- Boucle `for i in range(n)` : à chaque étape, 4 opérations élémentaires (appel de l'élément `L[i]`, affectation, appel de l'élément `HISTO[valeur]`, incrément de 1).

  On a donc $4n$ opérations élémentaires.
- Boucle `for i in range(p+1)` : à chaque étape, on a un test avec 2 opérations élémentaires (appel de l'élément `HISTO[i]` et test).

  L'ajout d'un élément dans `L_tri` se fait au maximum n fois lorsque toutes les étapes de la boucle `for i in range(p+1)` ont été effectuées. Lorsqu'on ajoute un élément dans `L_tri`, on a 3 opérations élémentaires (appel de l'élément `HISTO[i]`, fonction `int`, ajout de i dans `L_tri`). On a donc $2(p+1) + 3n$ opérations élémentaires pour la boucle `for i in range(p+1)`.

Le nombre total d'opérations élémentaires est : $2+1+4n+(p+1)+4n+2(p+1)+3n$.

La complexité est linéaire en $O(n+p)$.

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

4.

```
import matplotlib.pyplot as plt
  # module matplotlib.pyplot renommé plt
plt.figure()                          # nouvelle fenêtre graphique
plt.hist(L, range=(10, 20), bins=10)
                                      # tracé de l'histogramme
                                      # minimum = 10 et maximum = 20
                                      # avec 10 intervalles
plt.title('Histogramme de la liste L') # titre de l'histogramme
plt.xlabel('Valeurs de L')
plt.ylabel("Nombre d'occurrences")
plt.show()                            # affiche la figure à l'écran
```

5. On peut réduire le nombre d'éléments 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 `maxi+1` éléments dans la question 2.

```
def tri_comptage2(L):
    # la fonction retourne L_tri, qui est la liste L triée par
    # ordre croissant
    n=len(L)                    # nombre d'éléments de L
    L_tri=[]
    mini, maxi=L[0], L[0]       # recherche du minimum
                                # et du 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)
```

Exercice 10.6 : 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…

1. Écrire une fonction `tri_bulles` réalisant cette opération.
2. Donner les caractéristiques du tri à bulles.
3. Évaluer la complexité dans le pire des cas lors de l'appel de la fonction `tri_bulles`.
4. Proposer une amélioration de la fonction `tri_bulles` en tenant compte qu'aucun élément n'est échangé lors d'un parcours d'une liste triée.

Analyse du problème

On étudie dans cet exercice le tri à bulles. On parcourt la liste `L` en comparant les éléments consécutifs deux à deux et en faisant remonter vers la fin de la liste les plus grands éléments. Au bout du premier parcours, l'élément le plus grand est remonté comme une bulle, d'où le nom « tri à bulles ».

1. 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)`, dans laquelle 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 de 1 inclus à 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]`.

```
def tri_bulles(L):
    # la fonction trie par ordre croissant la liste L
    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=[8, 4, 2, 22, 6]
print(L)
tri_bulles(L)
print(L)
```

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.

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 (voir exercice 1.5 « Passage par référence pour les listes, effet de bord » dans le chapitre « Prise en main de Python »).

La variable `L` dans la fonction `tri_bulles` est une liste, qui est un objet muable. Il est donc inutile de retourner la variable `L` dans cette fonction !

2. Le tri à bulles est comparatif et itératif.

La liste initiale est triée par ordre alphabétique : `[[67, 'France'], [40, 'Irak'], [47, 'Kenya'], [32, 'Pérou'], [66, 'Royaume-Uni'], [66, 'Thaïlande']]`.

La liste triée par ordre croissant de la population est : `[[32, 'Pérou'], [40, 'Irak'], [47, 'Kenya'], [66, 'Royaume-Uni'], [66, 'Thaïlande'], [67, 'France']]`.

Pour trier la liste précédente, il suffit de remplacer la ligne `if L[j-1]>L[j]` par `if L[j-1][0]>L[j][0]`.

Le tri à bulles est donc stable puisqu'on garde l'ordre alphabétique dans la liste triée pour les pays qui ont la même population.

Le tri à bulles est un tri en place car il n'utilise pas de liste auxiliaire.

3. On cherche à calculer le nombre d'opérations élémentaires :

- Ligne `n=len(L)` : 2 opérations élémentaires (appel de `len(L)` et affectation).
- Boucle `for i in range(n-1, 0, -1)` :
  - Boucle `for j in range(1, i+1)` : on se place dans le pire des cas. On a 7 opérations élémentaires (appel de l'élément `L[j]`, appel de l'élément `L[j-1]`, test, appel de l'élément `L[j]`, appel de l'élément `L[j-1]`, 2 affectations).

Le nombre total d'opérations élémentaires vaut : $2+\sum_{i=1}^{n-1} 7i = \frac{7}{2}n^2 - \frac{7}{2}n + 2$.

La complexité est quadratique en $O(n^2)$.

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

4. Si aucun élément n'est échangé lors d'un parcours d'indice *i*, alors la liste est bien triée.

On ajoute une variable `liste_triée` qui passe à `False` si des éléments de la liste sont échangés. Dans ce cas, la liste n'est pas encore triée pour ce parcours d'indice *i*.

```
def tri_bulles2(L):
    # la fonction trie par ordre croissant la liste L
    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
```

Remarque : L'instruction `break` fait sortir de la boucle `for i in range(n-1, 0, -1)` et passe à l'instruction suivante (`# return L est inutile car L est passé en référence`). Comme il n'y a pas d'instruction après ce commentaire, on sort de la fonction `tri_bulles2`.

L'instruction `break` fait sortir d'une boucle `while` ou `for` et passe à l'instruction suivante alors que l'instruction `return` quitte la fonction.

Partie 8

Dictionnaire, pile, file, deque

Plan

Dictionnaire, pile, file, deque

11

Exercice 11.1 : Opérations de base sur les dictionnaires

On considère des opérations de base sur les dictionnaires.

1. Écrire une fonction `dico_vide` qui renvoie un dictionnaire vide.

2. Écrire une fonction `ajout_cle` qui admet comme arguments un dictionnaire, une clé et une valeur. Cette fonction ajoute le couple (`clé`, `valeur`) au dictionnaire.

3. Écrire une fonction `supp_cle` qui admet comme arguments un dictionnaire et une clé. Cette fonction supprime le couple (`clé`, `valeur`) correspondant à `clé`.

Analyse du problème

Les dictionnaires sont très souvent utilisés en informatique. Chaque élément du dictionnaire a une clé unique. Les éléments du dictionnaire ne sont pas ordonnés.

Cours :

Rappels sur les listes et les tuples

Les éléments d'une liste ou d'un tuple sont ordonnés. Pour récupérer un élément, on utilise un indice.

```
L1=["MPSI", "PTSI"] # liste (objet modifiable) contenant 2 éléments
print(L1[1])        # affiche "PTSI", d'indice 1 dans la liste L1
L2=(48, 46)         # tuple (objet non modifiable) contenant 2 éléments
print(L2[0])        # affiche 48, d'indice 0 dans la liste L2
```

Dictionnaires

Une table de hachage est une structure de données permettant de stocker des couples (`clé`, `valeur`). Elle permet de retrouver une clé très rapidement. Les tables de hachage sont appelées dictionnaires avec Python. Dans un dictionnaire, on associe une valeur à une clé.

Le type de la clé peut être un entier, un nombre flottant, une chaîne de caractères… mais pas une liste.

Le type de la valeur associée à la clé peut être quelconque.

Les éléments d'un dictionnaire ne sont pas ordonnés. On ne peut pas utiliser un indice comme pour les listes pour accéder à un élément.

Chaque élément du dictionnaire est identifié par une clé unique.

On utilise la clé pour rechercher la valeur correspondante du couple (`clé`, `valeur`).

On définit un élément du dictionnaire dans Python en précisant la clé, suivie de « : » et de la valeur associée.

```
d1={"MPSI":48}      # dictionnaire dico1 constitué d'un seul élément
print(d1)           # affiche {'MPSI': 48}. On visualise la clé
                    # et la valeur
print(type(d1))     # affiche le type de d1 : dict (type dictionnaire)
```

`d1` est un objet de type `dict`.

On crée le dictionnaire `d2` constitué de deux éléments :

```
d2={"MPSI":48,"PTSI":46}
```

Les éléments sont délimités par des accolades.

Création d'un dictionnaire vide

```
d={}                # dictionnaire vide avec les accolades {}
```

Ajout d'un élément dans le dictionnaire

```
d["MPSI"]=48        # ajoute "MPSI" : 48
d["MP"]=46          # ajoute "MP" : 46
print(d)            # affiche {'MPSI': 48, 'MP': 46}
```

La clé est unique dans un dictionnaire. On ne peut pas ajouter "MPSI" : 45 dans `d`.

Par contre, on peut modifier une valeur :

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

Suppression d'un élément

```
del d["MPSI"]
print (d)           # {'MP': 46}
d["MPSI"]=48
d["PCSI"]=48
d["PTSI"]=46
print (d)           # {'MP': 46, 'MPSI': 48, 'PCSI': 48, 'PTSI': 46}
```

Teste si une clé est dans un dictionnaire

```
print("PCSI" in d)  # affiche True
```

Nombre d'éléments d'un dictionnaire

```
print("Nombre d'éléments de d :", len(d))
    # Python affiche : Nombre d'éléments de d : 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

```
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)
```

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`.

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

Les dictionnaires sont des objets muables (voir exercice 1.4 « Affectation, objet immuable, copie » dans le chapitre « Prise en main de Python »).

Remarque : On rencontre deux catégories 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 a 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.

1.

```
def dico_vide():   # la fonction renvoie un dictionnaire vide
    return {}
```

2.

```
def ajout_cle(dico, clé, valeur):
    # la fonction ajoute le couple (clé, valeur) à dico
    dico[clé]=valeur
```

3.

```
def supp_cle(dico, clé):
    # la fonction supprime le couple (clé, valeur)
    # correspondant à clé pour dico
    del dico[clé]
```

Remarque :

Le programme suivant permet de tester les fonctions précédentes :

```
d={}
print(d)
ajout_cle(d,"MPSI",48)
ajout_cle(d,"MP",46)
ajout_cle(d,"PCSI1",48)
ajout_cle(d,"PCSI2",48)
print(d)
print()
supp_cle(d,"MPSI")
print(d)
```

Exercice 11.2 : Comptage des éléments d'une liste à l'aide d'un dictionnaire

On considère la liste : `L=[10, 12, 10, 8, 6, 10, 12, -5, 8.2, 8.2]`.

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

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

Analyse du problème

Les éléments du dictionnaire ne sont pas ordonnés. Chaque élément du dictionnaire a une clé unique. La valeur de la clé est égale au nombre d'occurrences de la clé dans la liste.

1.

```
def comptagedico(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
    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 clé elt au dictionnaire
                            # elt apparaît la première fois dans d
                            # la valeur de la clé elt vaut 1
    return d
```

2.

```
L=[10, 12, 10, 8, 6, 10, 12, -5, 8.2, 8.2]
d=comptagedico(L)
print(d)                    # affichage du dictionnaire
```

Le programme Python affiche : `{10.0: 3, 12.0: 2, 8.0: 1, 6.0: 1, -5.0: 1, 8.2: 2}`.

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

```
mot="C'est un mot"
d=comptagedico(mot)
print(d)
```

Le programme Python affiche : `{'C': 1, "'": 1, 'e': 1, 's': 1, 't': 2, ' ': 2, 'u': 1, 'n': 1, 'm': 1, 'o': 1}`.

L'avantage d'utiliser un dictionnaire pour stocker le nombre d'occurrences est qu'il n'est pas nécessaire de connaître à l'avance les éléments de `mot`. On n'utilise de la place mémoire que pour les caractères qui apparaissent réellement dans `mot`.

Exercice 11.3 : Opérations de base sur les piles

On considère trois opérations de base sur les piles. On modélise une pile avec une liste `P` dont on ne peut ajouter et supprimer un élément qu'à une extrémité appelée sommet de pile (ou tête de pile). On utilise `P=[]` pour créer une pile vide.

1. Écrire une fonction `empiler` qui admet comme arguments une pile `P` et un élément `x`. Cette fonction ajoute l'élément `x` au sommet de la pile `P`.

2. Écrire une fonction `depiler` qui admet comme argument une pile `P` non vide. Cette fonction supprime le dernier élément entré dans la pile et retourne cet élément.

3. Écrire une fonction `pile_vide` qui admet comme argument une pile `P`. Cette fonction retourne `True` si la pile est vide et `False` sinon.

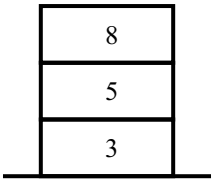
Analyse du problème

Les piles sont très souvent utilisées en informatique (voir chapitre 12 « Graphes »). On étudie dans ce chapitre une modélisation des piles avec des listes. Toutes les opérations sur la pile sont effectuées sur la même extrémité : on utilise le principe LIFO (Last In, First Out).

Cours :

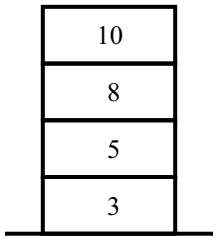
Une pile est une structure de données qui utilise le principe LIFO (Last In, First Out : « dernier entré, premier sorti »). On peut comprendre le fonctionnement d'une pile en considérant une pile d'assiettes.

- La fonction `empiler` consiste à ajouter une assiette sur le sommet de la pile (ou tête de la pile).
  Soit une pile `P` contenant 3 éléments : 3, 5 et 8.

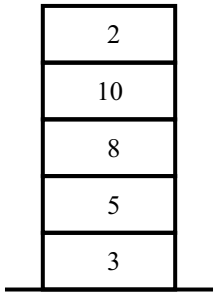


pile P

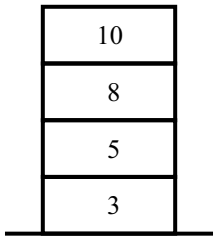
On souhaite empiler l'élément 10 à la pile `P`. On obtient alors la pile suivante :



Si on empile l'élément 2, on obtient :



- La fonction `depiler` consiste à supprimer un élément de la pile. L'élément à supprimer est toujours situé au sommet de la pile. On a bien une structure LIFO puisque le dernier élément rentré est le premier sorti.
  On obtient alors la pile :



La fonction « Undo » (Annulation de la frappe) des traitements de texte utilise une pile.

Remarque : L'ajout et la suppression d'un élément en fin de liste Python est très rapide. L'utilisation des listes Python pour gérer des piles est très efficace.

1.

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

Si on modifie un argument d'entrée muable (liste, dictionnaire, deque) dans une fonction alors on ne retrouve pas l'état initial de l'objet lorsqu'on quitte la fonction (voir exercice 1.5 « Passage par référence pour les listes, effet de bord » dans le chapitre « Prise en main de Python »).

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

2.

```
def depiler(P):
    # la fonction supprime le dernier élément entré dans la pile P
    # et retourne cet élément
    x=P.pop()    # supprime le dernier élément de la liste P
    return x     # retourne la valeur du dernier élément
                 # de la liste P
```

3.

```
def pile_vide(P):
    # la fonction retourne True si la pile P est vide
    # et False sinon
    if P==[]:   # on pourrait écrire : if len(P)==0:
        return True
    else:
        return False
```

Remarque :

Le programme suivant permet de visualiser les étapes du rappel de cours précédent :

```
P1=[]   # création d'une liste vide
print(pile_vide(P1))
empiler(P1,3)
empiler(P1,5)
empiler(P1,8)
empiler(P1,10)
empiler(P1,2)
print(P1)
y=depiler(P1)
print(y)
print(P1)
print(pile_vide(P1))
```

Exercice 11.4 : Parenthésage

On cherche à vérifier si une chaîne de caractères `L` est bien parenthésée, c'est-à-dire si le nombre ainsi que l'ordre des parenthèses ouvrantes et fermantes sont corrects. On note « (» la parenthèse ouvrante et «) » la parenthèse fermante.

On utilisera exclusivement les trois fonctions décrites dans l'exercice 11.3 « Opérations de base sur les piles » : `empiler`, `depiler` et `pile_vide` ainsi que `P=[]` pour créer une pile `P`.

Écrire une fonction `parenthesage` qui admet comme argument une chaîne de caractères `L`. Cette fonction retourne `True` si la chaîne de caractères est bien parenthésée, `False` sinon.

Exemples :

- `L='(2+8)/(5+9)'`

  `parenthesage(L)` doit retourner `True`.
- `L='(2+8)/((5+9)'`

  `parenthesage(L)` doit retourner `False`.

Analyse du problème

La structure de piles est parfaitement adaptée à la résolution de cet exercice. On parcourt les différents caractères de la chaîne `L`. On empile les parenthèses ouvrantes et on dépile dès qu'on a une parenthèse fermante.

On définit une pile `P` initialement vide. On parcourt tous les caractères de la chaîne `L`.

Dès qu'on rencontre une parenthèse ouvrante, on empile le caractère « (» dans `P`.

Quand on rencontre une parenthèse fermante, plusieurs cas interviennent :

- Si la pile `P` est vide, alors la fonction `parenthesage` retourne `False` puisqu'il manque une parenthèse ouvrante avant la parenthèse fermante.
- Sinon, on dépile la parenthèse ouvrante de `P`.

Lorsqu'on a parcouru tous les caractères de `L`, on doit avoir rencontré autant de parenthèses ouvrantes que fermantes. La pile `P` est nécessairement vide. Si ce n'est pas le cas, la fonction `parenthesage` retourne `False`.

```
def parenthesage(L):
    P=[]   # initialisation de la pile
    for elt in L:
        # on parcourt tous les caractères de L
        if elt=="(":   # parenthèse ouvrante empilée dans P
            empiler(P,elt)
        elif elt==")":   # parenthèse fermante
            if pile_vide(P)==True:
                # la pile ne doit pas être vide
```

```
                # il manque une parenthèse ouvrante
                return(False)
            else:
                depiler(P)
                # on dépile la parenthèse ouvrante

    if pile_vide(P)==True:
        return True
    else:
        return False
```

Exercice 11.5 : Opérations de base sur les files

On considère trois opérations de base sur les files. On modélise une file avec une liste `F` dont on ne peut ajouter un élément qu'à une extrémité et supprimer un élément qu'à l'autre extrémité. On rappelle que `F.pop(0)` permet de supprimer `F[0]` dans la liste `F`.

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

2. Écrire une fonction `défiler` qui admet comme argument une file `F` non vide. Cette fonction supprime le premier élément entré dans la file et retourne cet élément.

3. Écrire une fonction `file_vide` qui admet come argument une file `F`. Cette fonction retourne `True` si la file est vide et `False` sinon.

4. On considère le programme suivant :

```
F=[3, 5, 8]
enfiler(F, 10)
print(F)
enfiler(F, 2)
print(F)
x=défiler(F)
print('F :', F, 'x :',x)
print(file_vide(F))
```

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

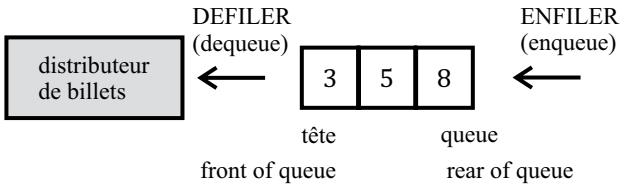
Analyse du problème

Les files sont très souvent utilisées en informatique (voir chapitre 12 « Graphes »). On étudie une modélisation des files avec des listes. On utilise le principe FIFO (First In, First Out).

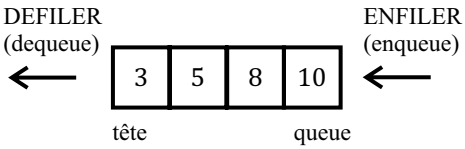
Cours :
Une file (queue en anglais) est une structure de données qui utilise le principe FIFO (First In, First Out : « premier entré, premier sorti »).

Dans une file d'attente à un distributeur de billets, les personnes font la queue les unes derrière les autres. Le premier arrivé dans la queue est le premier sorti (c'est-à-dire le premier servi pour obtenir les billets).

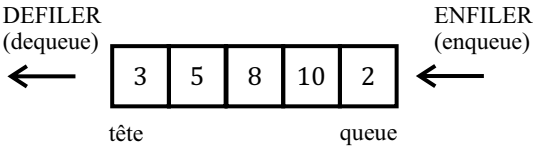
- La fonction `enfiler` (enqueue) consiste à ajouter un élément à la queue de la file (on dit aussi à l'arrière de la file d'attente).
  Soit une file d'attente `F` contenant 3 éléments :



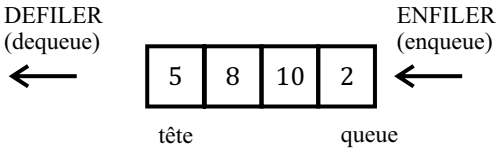
  On enfile l'élément 10 à la file `F`. On obtient alors :



  On enfile l'élément 2, on obtient :



  On modélise la file d'attente par une liste Python : `F = [3, 5, 8, 10, 2]`. On dit que 3 est à la tête de la file `F` et 2 est à la queue de la file `F`.
- La fonction `défiler` (dequeue) consiste à supprimer l'élément situé à la tête de la file (on dit aussi au début de la file d'attente). On a bien une structure FIFO puisque le premier élément rentré est le premier sorti.
  On obtient alors la file :



  La liste `F` s'écrit : `F = [5, 8, 10, 2]`.

Remarque : La suppression d'un élément en tête de liste Python n'est pas très rapide puisque tous les autres éléments doivent être décalés d'une position. On utilisera dans l'exercice suivant, « Utilisation des deques », la classe `collections.deque` qui est conçue pour ajouter et supprimer rapidement des éléments aux deux extrémités.

1.

```
def enfiler(F, x):    # F est une liste Python
    F.append(x)       # ajoute x à la queue de la file ou à la fin
                      # de la liste F
```

Si on modifie un argument d'entrée muable (liste, dictionnaire, deque) dans une fonction alors on ne retrouve pas l'état initial de l'objet lorsqu'on quitte la fonction (voir exercice 1.5 « Passage par référence pour les listes, effet de bord » dans le chapitre « Prise en main de Python »).

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

2.

```
def défiler(F):  # F est une liste Python
    x=F.pop(0)   # supprime l'élément situé à la tête de la file F
                 # c'est le premier élément de la liste F
    return x     # retourne x
```

3.

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

4. Le programme Python affiche :

```
[3, 5, 8, 10]
[3, 5, 8, 10, 2]
F : [5, 8, 10, 2] x : 3
False
```

Exercice 11.6 : 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.

1. Écrire une fonction `insere_gauche_deque` d'arguments une deque `D` et un élément *x* permettant d'ajouter *x* à l'extrémité gauche de la deque.

2. Écrire une fonction `insere_droite_deque` d'arguments une deque `D` et un élément *x* permettant d'ajouter *x* à l'extrémité droite de la deque.

3. On considère le programme suivant :

```
L1=[8, 3, 5]
D=deque()
for elt in L1:
    insere_gauche_deque(D, elt)
print(D)
L2=[10, 13, 1]
for elt in L2:
    insere_droite_deque(D, elt)
print(D)
E=deque([3, 2])
print(len(E)==0)
```

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

Analyse du problème

On considère les deques (double-ended queue), qui sont une généralisation des piles et des files. Les deques permettent d'ajouter et de supprimer très rapidement des éléments aux deux extrémités. On utilisera les deques dans le parcours des graphes.

Cours :

Une deque (se prononce « dèque ») est une structure de données qui généralise le fonctionnement des piles et des files. On peut ajouter et supprimer des éléments aux deux extrémités.

```
from collections import deque    # module permettant d'utiliser
                                 # les deques
```

Pour créer une deque vide :

```
D=deque()        # création d'une deque vide
```

On peut ajouter des éléments aux extrémités droite et gauche de la deque.

```
D.append(3)      # ajoute un élément à l'extrémité droite de D
D.append(5)      # ajoute un élément à l'extrémité droite de D
D.appendleft(8)  # ajoute un élément à l'extrémité gauche de D
D.appendleft(10) # ajoute un élément à l'extrémité gauche de D
```

On obtient alors la deque suivante :

10	8	3	5

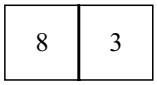
On a un nouveau type : `deque`. Ne pas confondre avec le type `list`.

```
print(type(D))   # le type de D est : deque
```

On peut supprimer des éléments à l'extrémité gauche ou droite de la deque.

```
x=D.pop()      # supprime et renvoie 5, l'élément à l'extrémité droite
               # de la deque
y=D.popleft() # supprime et renvoie 10, l'élément à l'extrémité gauche
               # de la deque
```

On obtient alors la deque suivante :




```
print('Teste si D est vide : ', len(D)==0) # teste si la deque est vide
```

On obtient sur l'afficheur : `False`

```
F=deque()      # création d'une deque vide
print('Teste si F est vide : ', len(F)==0) # teste si la deque est vide
```

On obtient sur l'afficheur : `True`

Dans Python, les deques (comme les listes et les dictionnaires) sont des objets muables.

```
E=D
```

Si on modifie `E`, alors `D` est également modifié puisque `D` et `E` font référence à la même adresse mémoire (voir exercice 1.4 « Affectation, objet immuable, copie » dans le chapitre « Prise en main de Python »).

```
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 superficielle de `D`, il ne faut pas écrire `E=D` mais utiliser la fonction `copy`.

```
import copy
E=copy.copy(D)  # copie superficielle de D
E.pop()         # supprime l'élément à l'extrémité droite de la deque E
# D n'est pas modifié puisque D ne pointe pas vers la même
# adresse mémoire que E
```

1.

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

2.

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

3. Le programme Python affiche :

```
deque([5, 3, 8])
deque([5, 3, 8, 10, 13, 1])
False
```

La deque `E` vaut : 3, 2. L'afficheur retourne `False` puisque la deque `E` n'est pas vide.

Remarques :

L'instruction `D=deque(3)` n'est pas correcte et renvoie un message d'erreur.

`D=deque([3])` définit la deque valant 3.

`D=deque('ijk')` définit la deque valant 'i', 'j', 'k'.

`D=deque(['ijk'])` définit la deque valant 'ijk'.

`D=deque([['abc'], ['ijk']])` définit la deque valant ['abc'], ['ijk'].

`x=D.popleft()` permet de supprimer ['abc'] et d'obtenir `x = ['abc']`.

Partie 9

Graphes

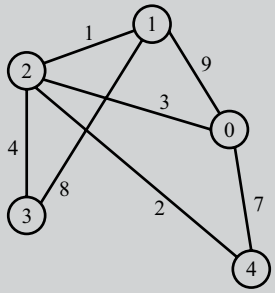
Plan

Graphes

12

Exercice 12.1 : Matrice d'adjacence

On considère le graphe $G = (S, A)$ non orienté, où le nombre situé sur l'arête joignant deux sommets est leur distance, supposée entière :



On utilise les listes de listes pour représenter les matrices dans Python.

1. Construire la matrice d'adjacence $\left(\mathbf{M}_{i,j}\right)_{0\leq i,j\leq 4}$ (appelée également **matrice de distance**) du graphe G, définie par :

Pour tous les indices i, j, $\mathbf{M}_{i,j}$ représente la distance entre les sommets i et j, ou encore la longueur de l'arête reliant les sommets i et j.

On convient que, lorsque les sommets ne sont pas reliés, cette distance vaut l'infini. On définit la variable `inf=1e10` qui représente une distance infinie. Écrire la matrice **M**.

2. Écrire une fonction `voisins`, d'arguments une matrice d'adjacence `M`, un sommet `i`, renvoyant la liste des voisins du sommet `i`.

3. Écrire une fonction `degré`, d'arguments une matrice d'adjacence `M`, un sommet `i`, renvoyant le nombre de voisins du sommet `i`, c'est-à-dire le nombre d'arêtes issues de `i`.

4. Écrire une fonction `longueur`, d'arguments une matrice d'adjacence `M`, une liste `L` de sommets de G, renvoyant la longueur du trajet décrit par la liste `L`, c'est-à-dire la somme des longueurs des arêtes empruntées. Si le trajet n'est pas possible, la fonction renverra -1.

Analyse du problème

On définit une matrice d'adjacence contenant les distances entre les différents sommets. Si deux sommets i et j ne sont pas reliés, alors $\mathbf{M}[i][j] = \text{inf}$. Cet exercice est extrait du concours banque PT 2015 Sujet 0. Comme le graphe n'est pas orienté, la matrice est symétrique : $\mathbf{M}[i][j] = \mathbf{M}[j][i]$.

Cours :

Un **graphe** G est un schéma contenant des points appelés **sommets** (ou **nœuds** ou points), reliés ou non par des arêtes (ou segments ou liens ou lignes).

On utilise la notation suivante : $\boldsymbol{G = (S, A)}$ est un couple d'ensemble finis, dont :

- S est l'ensemble des sommets de G ;
- A est l'ensemble des arêtes de G.

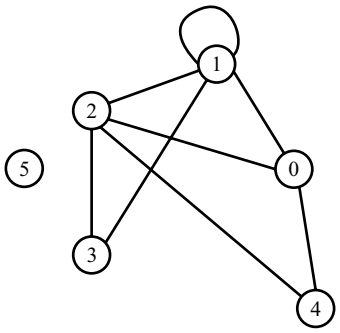
Si une arête relie les sommets s et s', on dit que **les sommets s et s' sont voisins ou adjacents**.

L'**ordre** d'un graphe est le nombre total de sommets.

Un graphe est orienté si les arêtes sont orientées, c'est-à-dire si on ne peut les parcourir que dans un sens.

Pour les graphes non orientés :

- Deux sommets sont **adjacents** lorsqu'ils sont reliés par une **arête**.
- La **taille** d'un graphe non orienté est le nombre total d'arêtes.
- Le **degré** d'un sommet s, noté $d(s)$ est égal au nombre d'arêtes dont ce sommet est une extrémité. Une **boucle** est une arête reliant un sommet à lui-même. Les boucles sont comptées deux fois.



  Le degré du sommet 2 vaut 4. Le degré du sommet 1 vaut 5. Le degré du sommet 5 vaut 0 : $d(5) = 0$.
  On n'étudiera par la suite que des graphes sans boucle.
- Une **chaîne** reliant un sommet s à un sommet s' est une suite d'arêtes consécutives permettant de se rendre de s à s'.
- La **longueur d'une chaîne** est le nombre d'arêtes de la chaîne, ou la somme des poids des arêtes qui le constituent.
- La **distance** $dist(s, s')$ entre deux sommets s et s' est la longueur d'une 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 l**ongueur d'un chemin** est le nombre d'arcs du chemin, ou la somme des poids des arcs qui le constituent.
- La **distance** $dist(s,s')$ entre deux sommets s et s' est la longueur 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 de s à s' et aussi un chemin de s' à s. Il n'y a pas de sommet isolé.
- Le poids des arcs peut être négatif. Un **cycle de poids 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.

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 $\mathbf{M}[i][j] = \mathbf{M}[j][i] = 1$ sinon $\mathbf{M}[i][j] = \mathbf{M}[j][i] = 0$.
- Graphe orienté : S'il existe un arc allant de i vers j, alors $\mathbf{M}[i][j] = 1$, sinon $\mathbf{M}[i][j] = 0$.

2) Graphe pondéré :

- Graphe non orienté : Si deux sommets différents i et j sont reliés par une arête, alors $\mathbf{M}[i][j] = \mathbf{M}[j][i]$ = distance entre les sommets i et j, sinon $\mathbf{M}[i][j] = \mathbf{M}[j][i] = \text{inf}$.
- Graphe orienté : S'il existe un arc allant de i vers j, alors $\mathbf{M}[i][j]$ = distance entre les sommets d'origine i et d'extrémité j, sinon cette distance vaut l'infini : $\mathbf{M}[i][j] = \text{inf}$. Le poids des arcs peut être négatif (voir exercice 14.4 « Algorithme de Floyd-Warshall » dans le chapitre « Programmation dynamique »).

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.

Algorithmes de parcours de graphe :

On va étudier plusieurs algorithmes de parcours de graphe : parcours en largeur (avec une file FIFO – First In, First Out : « premier entré, premier sorti »), parcours en profondeur (avec une pile LIFO – Last In, First Out : « dernier entré, premier sorti »).

Un graphe non orienté, connexe et acyclique est un arbre.

Chaque élément de l'arbre est appelé un nœud. Au niveau élevé, on trouve le nœud racine.

Au niveau juste en dessous, on trouve les nœuds fils (ou descendants). Un nœud n'ayant aucun fils est appelé feuille.

Le nombre de niveaux de l'arbre est appelé hauteur.

On peut construire un graphe à partir d'un autre. Un sous-graphe G' d'un graphe G est composé de certains des sommets de G et de certaines des arêtes de G.

Un graphe H est un **arbre couvrant** du graphe G si H est un arbre que l'on peut obtenir en supprimant des arêtes de G.

Remarque : On peut définir une matrice d'incidence $n \times p$ où n désigne le nombre de sommets du graphe et p le nombre d'arêtes (ou d'arcs).

Les exemples suivants peuvent être modélisés par des graphes :

- Site web : chaque page est un sommet du graphe. Un lien hypertexte est une arête entre deux sommets.
- Réseau ferroviaire : chaque gare est un sommet. Les voies entre deux gares sont des arêtes.
- Réseau routier : chaque ville est un sommet. Les routes entre deux villes sont des arêtes.
- Réseau social : les sommets sont des personnes. Deux personnes sont adjacentes lorsqu'elles sont amies. Le graphe est orienté si l'amitié n'est pas réciproque entre deux personnes.

1. La matrice d'adjacence est : $\mathbf{M} = \begin{pmatrix} 0 & 9 & 3 & \infty & 7 \\ 9 & 0 & 1 & 8 & \infty \\ 3 & 1 & 0 & 4 & 2 \\ \infty & 8 & 4 & 0 & \infty \\ 7 & \infty & 2 & \infty & 0 \end{pmatrix}$.

Remarque :

Tous les éléments de la diagonale sont nuls puisque la distance entre les sommets i et i est nulle : $\mathbf{M}[i][j] = 0$.

Les éléments sont symétriques par rapport à la diagonale puisque la distance entre les sommets i et j est la même qu'entre les sommets j et i : $\mathbf{M}[i][j] = \mathbf{M}[j][i]$.

On peut déduire de la deuxième ligne de la matrice que le sommet 1 est relié aux sommets : 0, 2 et 3.

Pour la troisième ligne, le sommet 2 est relié aux sommets : 0, 1, 3 et 4.

```
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]]
```

2.

```
def voisins(M, i):
    # la fonction renvoie la liste des voisins du sommet i
    # pour la matrice M
    n=len(M)            # nb de lignes de la matrice d'adjacence
    L=[]                # initialisation de la liste L
    for j in range(n):  # j varie entre 0 inclus et n exclu
        if   M[i][j]>0 and M[i][j]<inf:
            L.append(j) # si 0 < distance < inf, on ajoute le
                        # sommet dans L
    return (L)
```

On obtient par exemple `voisins(M, 4)` = [0,2].

3.

```
def degré(M, i):
    # la fonction renvoie le nombre de voisins du sommet i
    # pour la matrice M
    n=len(M)           # nb de lignes de la matrice d'adjacence
    somme=0            # le nombre d'arêtes vaut 0
    for j in range(n): # j varie entre 0 inclus et n exclu
```

```
        if M[i][j]>0 and M[i][j]<inf:
            somme+=1    # incrémente de 1 le nombre d'arêtes
                        # si distance > 0
    return somme
```

On obtient par exemple `degré(2)` = 4. On écrit alors : $d(2) = 4$.

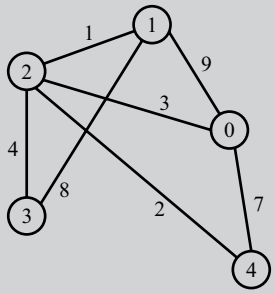
4.

```
def longueur(M, L):
    # la fonction renvoie la longueur du trajet décrit par
    # la liste L pour la matrice M
    somme=0
    n=len(L)
    for i in range(n-1): # i varie entre 0 inclus et n-1 exclu
        if M[L[i]][L[i+1]]>=0:
            somme+=M[L[i]][L[i+1]]
        else:
            somme=-1
            return somme # la fonction s'arrête et retourne -1
    return somme
```

On obtient par exemple : `longueur(M, [0,1,3,2])` = 21.

Exercice 12.2 : Graphe avec liste d'adjacence. Dictionnaire des sommets adjacents

On considère le graphe $G = (S, A)$ non orienté, où le nombre situé sur l'arête joignant deux sommets est leur distance, supposée entière :



1. Définir un dictionnaire représentant le graphe G. Chaque clé est associée à un sommet. La valeur de la clé représente le dictionnaire des sommets adjacents, dont la valeur de la clé est la distance entre les deux sommets.

2. Écrire une fonction `voisins_dict`, d'arguments un dictionnaire `dico`, un sommet `i`, renvoyant la liste des voisins du sommet `i`.

3. Écrire une fonction `degre_dict`, d'arguments un dictionnaire `dico`, un sommet `i`, renvoyant le nombre de voisins du sommet `i`, c'est-à-dire le nombre d'arêtes issues de `i`.

4. Écrire une fonction `longueur_dict`, d'arguments un dictionnaire `dico`, une liste `L` de sommets de *G*, renvoyant la longueur du trajet décrit par cette liste `L`, c'est-à-dire la somme des longueurs des arêtes empruntées. Si le trajet n'est pas possible, la fonction renvoie –1.

Analyse du problème

On définit un dictionnaire où chaque clé représente un sommet. Pour un sommet donné, la valeur de la clé est un dictionnaire qui contient l'ensemble des sommets adjacents avec les distances entre les deux sommets.

1. Le dictionnaire est :

```
dico={0:{1:9, 2:3, 4:7},\
      1:{0:9, 2:1, 3:8},\
      2:{0:3, 1:1, 3:4, 4:2},\
      3:{1:8, 2:4},\
      4:{0:7, 2:2} }
```

Voir exercice 11.1 « Opérations de base sur les dictionnaires » dans le chapitre « Dictionnaire, pile, file, deque » pour l'utilisation des dictionnaires.

On peut écrire également :

```
dico=dict()
dico[0]={1:9, 2:3, 4:7}      # ajoute la clé 0 dans dico
dico[1]={0:9, 2:1, 3:8}      # ajoute la clé 1 dans dico
dico[2]={0:3, 1:1, 3:4, 4:2} # ajoute la clé 2 dans dico
dico[3]={1:8, 2:4}           # ajoute la clé 3 dans dico
dico[4]={0:7, 2:2}           # ajoute la clé 4 dans dico
```

2.

```
def voisins_dict(dico, i):
    # la fonction renvoie la liste des voisins du sommet i
    # pour le dictionnaire dico
    L=[] # initialisation de la liste L
    if i in dico:  # teste si la clé i est dans le diction. dico
        for clé, valeur in dico[i].items():
                    # parcourt les couples (clé, valeur) de dico[i]
            L.append(clé)
    return L
```

On obtient par exemple `voisins_dict(dico, 4)` = [0, 2].

3.

```
def degre_dict(dico, i):
    # la fonction renvoie le nombre de voisins du sommet i
    # pour le dictionnaire dico
    somme=0
    if i in dico:  # teste si la clé i est dans le diction. dico
        for clé, valeur in dico[i].items():
                    # parcourt les couples (clé, valeur) de dico[i]
            somme+=1
    return somme
```

On obtient par exemple `degre_dict(dico, 2)` = 4. On écrit alors : $d(2) = 4$.

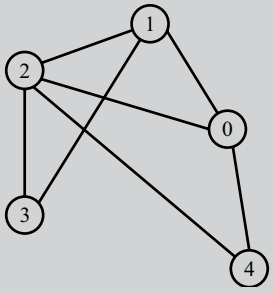
4.

```
def longueur_dict(dico, L):
    # la fonction renvoie la longueur du trajet décrit
    # par la liste L pour le dictionnaire dico
    somme=0
    n=len(L)
    for i in range(n-1): # i varie entre 0 inclus et n-1 exclu
        if L[i] in dico: # teste si la clé L[i] est dans dico
            dico2=dico[L[i]]
            if L[i+1] in dico2:
                somme+=dico2[L[i+1]]
            else:
                somme=-1
                return somme # la fonction s'arrête et retourne -1
        else:
            somme=-1
            return somme     # la fonction s'arrête et retourne -1
    return somme
```

On obtient par exemple : `longueur_dict(dico, [0, 1, 3, 2])` = 21.

Exercice 12.3 : Graphe avec liste d'adjacence. Liste des sommets adjacents

On considère le graphe $G = (S, A)$ non orienté :



1. Définir un dictionnaire représentant le graphe *G*. Chaque clé est associée à un sommet. La valeur de la clé représente la liste des sommets adjacents.

2. Écrire une fonction `voisins_dict`, d'arguments un dictionnaire `dico`, un sommet `i`, renvoyant la liste des voisins du sommet `i`.

3. Écrire une fonction `degre_dict`, d'arguments un dictionnaire `dico`, un sommet `i`, renvoyant le nombre de voisins du sommet `i`, c'est-à-dire le nombre d'arêtes issues de `i`.

Analyse du problème

On définit un dictionnaire où chaque clé représente un sommet. Pour un sommet donné, la valeur de la clé est une liste qui contient l'ensemble des sommets adjacents.

1. Pour chaque sommet, on écrit la liste des sommets accessibles.

Le dictionnaire est :

```
dico_liste={0:[1, 2, 4],\
            1:[0, 2, 3],\
            2:[0, 1, 3, 4],\
            3:[1, 2],\
            4:[0, 2]}
```

Le sommet 0 est relié aux sommets 1, 2 et 4. La valeur de la clé 0 est la liste des sommets adjacents [1, 2, 4].

Voir exercice 11.1 « Opérations de base sur les dictionnaires » dans le chapitre « Dictionnaire, pile, file, deque » pour l'utilisation des dictionnaires.

2.

```
def voisins_dict_liste(dico, i):
    # la fonction renvoie la liste des voisins du sommet i
    # pour le dictionnaire dico
    L=[]           # initialisation de la liste L
    if i in dico:  # teste si la clé i est dans le diction. dico
        for elt in dico[i]:
            L.append(elt)
    return L
```

On obtient par exemple `voisins_dict(dico, "4")` = [0, 2].

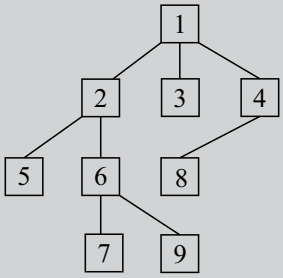
3.

```
def degre_dict_liste(dico, i):
    # la fonction renvoie le nombre de voisins du sommet i
    # pour le dictionnaire dico
    somme=0
    if i in dico:  # teste si la clé i est dans le diction. dico
        for elt in dico[i]:
            somme+=1
    return somme
```

On obtient par exemple `degre_dict(dico, 2)` = 4. On écrit alors : $d(2) = 4$.

Exercice 12.4 : Parcours en largeur d'un arbre en utilisant une file

On modélise un site web par une page d'accueil qui contient des liens hypertextes permettant d'accéder à d'autres pages du site qui peuvent contenir également des liens hypertextes. Le site web contient 9 pages numérotées de 1 à 9. On considère l'arbre $G = (S, A)$ représentant la structure du site web.



On utilisera les deux opérations de base sur les files : `defiler` pour la suppression d'un élément et `enfiler` pour l'ajout d'un élément. On rappelle que `F.pop(0)` permet de supprimer `F[0]` dans la liste `F`.

On cherche à parcourir en largeur tous les sommets de cet arbre *G* (toutes les pages web de ce site).

Les étapes de l'algorithme de parcours en largeur sont les suivantes :

- Ajouter le sommet de départ dans la file `F` initialement vide.
- Tant que la file `F` n'est pas vide, supprimer l'élément *x* à la tête de la file. Ajouter les fils de *x* dans la file.

1. Définir un dictionnaire `dico` représentant l'arbre *G*. Chaque clé est associée à un sommet *x*. La valeur de la clé représente la liste des fils du sommet *x*.

2. Écrire une fonction `parcourslargeur` qui admet comme arguments un dictionnaire `dico` et un sommet de départ `début` permettant de parcourir en largeur un arbre.

3. Dans quel ordre sont parcourus les sommets dans la fonction `parcourslargeur(dico, 1)` ?

Analyse du problème

L'algorithme de parcours en largeur (ou BFS, Breadth First Search, en anglais) permet de traiter les sommets adjacents à un sommet donné pour ensuite les explorer un par un.

L'implémentation repose sur une file `F` dans laquelle on place le premier sommet à la queue de `F` et les sommets adjacents non explorés à la queue de `F`. On utilise le principe FIFO (First In, First Out : « premier entré, premier sorti »).

Voir exercice 11.5 « Opérations de base sur les files » dans le chapitre « Dictionnaire, pile, file, deque » pour l'implémentation des files par les listes de Python en utilisant le principe FIFO.

Cours :

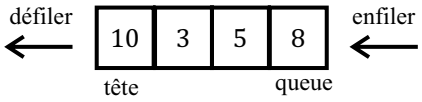
Un graphe non orienté connexe et acyclique est un arbre. Chaque élément de l'arbre est appelé un nœud.

Au niveau élevé, on trouve le nœud racine (1). Au niveau juste en dessous, on a trois nœuds fils (2, 3 et 4). Un nœud n'ayant aucun fils est appelé feuille. Les nœuds 3, 5, 7, 8 et 9 sont des feuilles.

Le nombre total de niveaux de l'arbre est appelé hauteur. La hauteur de l'arbre *G* vaut 4.

G est un arbre ternaire puisque chaque nœud comporte au plus trois fils au niveau inférieur. Du point de vue d'un fils, le nœud dont il est issu au niveau supérieur est appelé père.

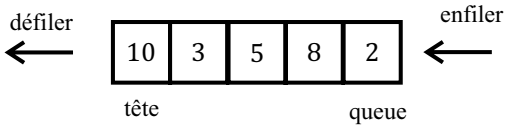
On utilise une liste `F` pour modéliser une file. Par exemple `F=[10, 3, 5, 8]`.



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.



1. Le premier niveau de l'arbre contient un sommet : 1.
Le deuxième niveau contient trois sommets : 2, 3 et 4.
Le troisième niveau contient trois sommets : 5, 6 et 8.
Le quatrième niveau contient deux sommets : 7 et 9.

```
dico={1:[2, 3, 4], 2:[5, 6], 3:[],4:[8],\
  5:[], 6:[7, 9], 7:[], 8:[],9:[]}
```

Le sommet 2 a deux fils : 5 et 6. La valeur de la clé 2 est la liste des fils : [5, 6].

2. On utilise dans Python les listes pour manipuler les files.
`F.pop(0)` permet de supprimer `F[0]` dans la liste `F`, c'est-à-dire l'élément situé à la tête de `F`.

Remarque : Ne pas confondre avec `F.pop()`, qui permet de supprimer le dernier élément de la liste `F`.

```
def enfiler(F, x):
    F.append(x)    # ajoute x à la queue de la file
                   # ou à la fin de la liste F

def defiler(F):
    x=F.pop(0)     # supprime l'élément situé à la tête de F : F[0]
    return x       # retourne x

def parcourslargeur(dico, début):
    # fonction permettant un parcours en largeur du dictionnaire
    # dico en partant de début
    F=[début]          # modélisation d'une file avec la liste F
```

```
while F!=[]:
    x=defiler(F)          # supprime l'élément à la tête de
                          # la file F : F[0]
    for elt in dico[x]:
        enfiler(F, elt)   # ajoute les fils du sommet x
                          # à la queue de la file F
```

Remarque :

`dico[x]` contient les fils du sommet *x*. *G* ne contient pas de cycle par définition d'un arbre. Tous les fils sont nécessairement non explorés. On peut donc tous les ajouter dans la file `F`.

On étudiera un graphe dans l'exercice suivant « Parcours en largeur d'un graphe avec une deque » : il faudra tester si les sommets adjacents ont déjà été explorés.

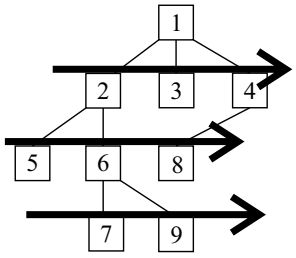
3. `F=[1]` au début de l'algorithme.

- On enlève 1 de la file `F` et on explore ce sommet. Les fils de « 1 » sont ajoutés au fur et à mesure à la queue de `F`. La file `F` vaut alors : `[2, 3, 4]`.
- On enlève 2, le premier élément de `F` (élément situé à la tête de la file). On explore le sommet 2. Les fils de « 2 » sont ajoutés à la queue de `F`. La file `F` vaut alors : `[3, 4, 5, 6]`.
- On enlève 3, le premier élément de `F` (élément situé à la tête de la file). On explore le sommet 3. Il n'y a pas de fils. La file `F` vaut alors : `[4, 5, 6]`.
- On enlève 4, le premier élément de `F` (élément situé à la tête de la file). On explore le sommet 4. Les fils de « 4 » sont ajoutés à la queue de `F`. La file `F` vaut alors : `[5, 6, 8]`.
- On enlève 5, le premier élément de `F` (élément situé à la tête de la file). On explore le sommet 5.
- ...

Finalement, on a exploré les sommets dans l'ordre :

- 1,
- 2, 3, 4,
- 5, 6, 8,
- 7, 9.

Les flèches sur le graphe représentent le sens de parcours niveau par niveau et de gauche à droite. On a bien exploré les sommets en largeur.



> On va étudier une amélioration du parcours en largeur pour un graphe non orienté, connexe et possédant des cycles dans l'exercice suivant « Parcours en largeur d'un graphe avec une deque ».

Remarques :

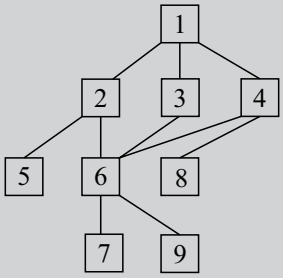
L'ordre de parcours des sommets pour un niveau donné n'a pas d'importance. Ici, on les parcourt dans le sens croissant, soit de gauche à droite.

Le dictionnaire `dico2` ci-dessous contient tous les sommets adjacents à un sommet donné. La fonction `parcourslargeur(G)` ne se termine jamais puisque la boucle `for` ajoute tous les sommets adjacents. Dans l'exercice suivant « Parcours en largeur d'un graphe avec une deque », on va résoudre ce problème en ne considérant que les sommets adjacents non explorés.

```
dico2={1:[2, 3, 4], 2:[1, 5, 6], 3:[1, 6],\
  4:[1, 6, 8], 5:[2], 6:[2, 7, 9], 7:[6],\
  8:[4], 9:[6]}
```

Exercice 12.5 : Parcours en largeur d'un graphe avec une deque

On modélise un site web par une page d'accueil qui contient des liens hypertextes permettant d'accéder à d'autres pages du site qui peuvent contenir également des liens hypertextes. Le site web contient 9 pages numérotées de 1 à 9. On considère le graphe non orienté $G = (S, A)$ représentant la structure du site web.



On cherche à parcourir en largeur tous les sommets de ce graphe (toutes les pages web de ce site). On utilise la liste `VISITED` représentant les sommets déjà explorés.

Les étapes de l'algorithme de parcours en largeur sont les suivantes :

- Ajouter le sommet de départ dans la deque `D` initialement vide.
- Tant que `D` n'est pas vide, supprimer le sommet x à l'extrémité gauche de `D`. Ajouter à l'extrémité droite de `D` les sommets non explorés adjacents au sommet x.

1. Définir un dictionnaire `dico` représentant le graphe G. La clé associée à chaque sommet représente la liste des sommets adjacents.

2. Écrire une fonction `parcourslargeur2` qui admet comme arguments un dictionnaire `dico` et un sommet de départ `début` permettant de parcourir en largeur un graphe non orienté, connexe.

3. Dans la fonction `parcourslargeur2(dico, 1)`, dans quel ordre sont parcourus les sommets ?

4. Calculer la complexité de cet algorithme dans le pire des cas.

Analyse du problème

L'algorithme de parcours en largeur (ou BFS, Breadth First Search, en anglais) permet de traiter les sommets adjacents à un sommet donné pour ensuite les explorer un par un.

L'implémentation repose sur une deque `D` dans laquelle on place le premier sommet à l'extrémité droite de `D` initialement vide et les sommets adjacents non explorés à l'extrémité droite de `D`. On utilise le principe FIFO (First In, First Out : « premier entré, premier sorti »). Voir exercice 11.6 « Utilisation des deques » dans le chapitre « Dictionnaire, pile, file, deque » pour la manipulation des deques.

Le graphe contient des cycles. Pour ne pas explorer plusieurs fois un même sommet, on marque les sommets déjà explorés.

1.

```
dico={1:[2, 3, 4], 2:[1, 5, 6],3:[1, 6], 4:[1, 6, 8],\
  5:[2], 6:[2, 7, 9],7:[6], 8:[4],9:[6]}
```

Le sommet 2 est relié aux sommets 1, 5 et 6. La valeur de la clé 2 est la liste des sommets adjacents [1, 5, 6].

2. Les sommets déjà visités sont marqués pour éviter d'explorer plusieurs fois un même sommet. La liste `VISITED` contient les sommets visités.

Les étapes de l'algorithme de parcours en largeur sont les suivantes :

Initialisation de l'algorithme :

- Mettre le sommet de départ dans la deque `D` initialement vide.
- La liste `VISITED` contient le sommet de départ : `VISITED=[début]`.

Boucle tant que la deque `D` n'est pas vide :

- Supprimer le sommet *x* à l'extrémité gauche de `D`.
- Ajouter dans `VISITED` et à l'extrémité droite de `D` les sommets non explorés adjacents au sommet *x*.

```
def parcourslargeur2(dico, début):
    # fonction permettant un parcours en largeur du dictionnaire
    # dico en partant de début et retournant
    # VISITED la liste des sommets visités
    from collections import deque # module permettant d'utiliser
                                  # les deques
```

```
D=deque()          # deque vide
D.append(début)    # ajoute le sommet de départ
VISITED=[début]
while len(D)!=0:
    x=D.popleft() # supprime le sommet x à l'extrémité
                  # gauche de D
    L=dico[x]     # liste contenant les sommets adjacents à x
    for elt in L: # parcourt les éléments de L
        if elt not in VISITED:  # teste si le sommet n'a pas
                                # déjà été exploré
            D.append(elt)       # ajoute le sommet elt à
                                # l'extrémité droite de D
            VISITED.append(elt) # le sommet elt a été exploré
```

Remarque :

`dico[x]` contient les sommets adjacents au sommet *x*. *G* peut contenir des cycles puisqu'on ne considère pas d'arbre dans cet exercice. Il faut tester si les sommets adjacents ont déjà été explorés.

Dans l'exercice précédent « Parcours en largeur d'un arbre en utilisant une file », on n'avait pas besoin de tester si les fils étaient déjà explorés puisqu'on considérait un arbre, sans cycle par définition.

3. La deque `D` vaut [1] au début de l'algorithme.

- On supprime l'élément à l'extrémité gauche de `D`. On explore ce sommet. Les sommets adjacents à 1 non explorés sont ajoutés à l'extrémité droite de `D`. La deque `D` vaut alors : [2, 3, 4].
- On supprime l'élément à l'extrémité gauche de `D`. On explore le sommet 2. Les sommets adjacents à 2 non explorés sont ajoutés à l'extrémité droite de `D`. La deque `D` vaut alors : [3, 4, 5, 6].
- On supprime l'élément à l'extrémité gauche de `D`. On explore le sommet 3. Le sommet 6 a déjà été exploré. On ne l'ajoute pas. La deque `D` vaut alors : [4, 5, 6].
- On supprime l'élément à l'extrémité gauche de `D`. On explore le sommet 4. Les sommets adjacents à 4 non explorés sont ajoutés à l'extrémité droite de `D`. La deque `D` vaut alors : [5, 6, 8].
- On supprime l'élément à l'extrémité gauche de `D`. On explore le sommet 5. Pas de nouveau sommet non exploré. La deque `D` vaut alors : [6, 8].
- On supprime l'élément à l'extrémité gauche de `D`. On explore le sommet 6. La deque `D` vaut alors : [8, 7, 9].
- On supprime l'élément à l'extrémité gauche de `D`. On explore le sommet 8. Pas de nouveau sommet non exploré. La deque `D` vaut alors : [7, 9].
- On supprime l'élément à l'extrémité gauche de `D`. On explore le sommet 7. Pas de nouveau sommet non exploré. La deque `D` vaut alors : [9].
- On supprime l'élément à l'extrémité gauche de `D`. On explore le sommet 9. Pas de nouveau sommet non exploré. La deque `D` est vide.

Avec la fonction `parcourslargeur2(dico, 1)`, on explore les sommets suivants : `VISITED = [1, 2, 3, 4, 5, 6, 8, 7, 9]`.

4. Soit *n* le nombre de sommets et *m* le nombre d'arêtes.

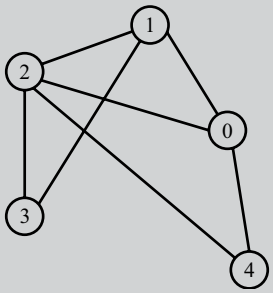
À chaque passage dans la boucle `while len(D)!=0`, on enlève un sommet à l'extrémité gauche de la deque `D`. Cette boucle sera donc exécutée au plus *n* fois.

À chaque fois que l'on supprime un sommet à l'extrémité gauche de la deque `D`, la boucle `for elt in L` parcourt tous les sommets adjacents au sommet supprimé. Comme il y a *m* arêtes, les lignes `D.append(elt)` et `VISITED.append(elt)` seront exécutées au plus une fois pour chaque arête, soit au plus *m* fois.

La complexité dans le pire des cas pour un graphe représenté par une liste d'adjacence est linéaire en $O(n + m)$.

Exercice 12.6 : Parcours en largeur d'un graphe avec une matrice d'adjacence

On considère le graphe non orienté $G = (S, A)$:



On utilise les listes de listes pour représenter les matrices dans Python.

1. Construire la matrice d'adjacence $\mathbf{M}_{i,j}$ du graphe *G*. Si deux sommets différents *i* et *j* sont reliés par une arête, alors $\mathbf{M}[i][j] = 1$ sinon $\mathbf{M}[i][j] = 0$.

2. Écrire une fonction `parcourslargeur_mat` d'arguments une matrice d'adjacence `M` et un sommet de départ `début` permettant de parcourir en largeur le graphe. On utilise une deque pour parcourir en largeur le graphe. La fonction affiche la liste des sommets explorés.

3. Dans quel ordre sont parcourus les sommets dans la fonction `parcourslargeur_mat(M, 0)` ?

4. Calculer la complexité de cet algorithme dans le pire des cas.

Analyse du problème

L'algorithme de parcours en largeur (ou BFS, Breadth First Search, en anglais) permet de traiter les sommets adjacents à un sommet donné pour ensuite les explorer un par un. Il utilise une deque `D` dans laquelle il place le premier sommet à l'extrémité droite

de D initialement vide et les sommets adjacents non explorés à l'extrémité droite de D. On utilise le principe FIFO (First In, First Out : « premier entré, premier sorti »).

1. La matrice d'adjacence est : $\mathbf{M} = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{pmatrix}$.

```
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 : $\mathbf{M}[i][i] = 0$.

Les éléments sont symétriques par rapport à la diagonale puisque le graphe n'est pas orienté : $\mathbf{M}[i][j] = \mathbf{M}[j][i]$.

On peut déduire de la deuxième ligne de la matrice que le sommet 1 est relié aux sommets : 0, 2 et 3.

Pour la troisième ligne, le sommet 2 est relié aux sommets : 0, 1, 3 et 4.

2. Les étapes de l'algorithme de parcours en largeur sont les suivantes :

Initialisation de l'algorithme :

- Mettre le sommet de départ dans la deque D initialement vide.
- La liste VISITED contient le sommet de départ : VISITED=[début].

Boucle tant que la deque D n'est pas vide :

- Supprimer le sommet *i* à l'extrémité gauche de D.
- Ajouter dans VISITED et à l'extrémité droite de D les sommets *j* non explorés adjacents au sommet *i*.

```
def parcourslargeur_mat(M, début):
    # fonction permettant un parcours en largeur de la matrice M
    # en partant de début et retournant VISITED la liste des
    # sommets visités
    from collections import deque # module permettant d'utiliser
                                  # les deques
    D=deque()                     # deque vide
    D.append(début)               # ajoute le sommet de départ
    VISITED=[début]               # liste des sommets explorés
    while len(D)!=0:
        i=D.popleft()             # supprime le sommet i à
                                  # l'extrémité gauche de D
        for j in range(len(M[i])):
            if M[i][j]==1 and (j not in VISITED):
                D.append(j)       # ajoute le sommet j à
                                  # l'extrémité droite de D
```

```
                VISITED.append(j) # le sommet j a été exploré
        print(D, VISITED)
    print('Sommets explorés :',VISITED)
```

3. La deque `D` vaut [0] au début de l'algorithme.

- On supprime l'élément à l'extrémité gauche de `D`. On explore le sommet 0. Les sommets adjacents à 0 non explorés sont ajoutés à l'extrémité droite de `D`. La deque `D` vaut alors : [1, 2, 4]. La liste `VISITED` vaut : [0, 1, 2, 4].
- On supprime l'élément à l'extrémité gauche de `D`. On explore le sommet 1. Les sommets adjacents à 1 et non explorés sont ajoutés à l'extrémité droite de `D`. La deque `D` vaut alors : [2, 4, 3]. La liste `VISITED` vaut : [0, 1, 2, 4, 3].
- On supprime l'élément à l'extrémité gauche de `D`. On explore le sommet 2. Pas de sommet non exploré adjacent à 2. La deque `D` vaut alors : [4, 3]. La liste `VISITED` vaut : [0, 1, 2, 4, 3].
- On supprime l'élément à l'extrémité gauche de `D`. Pas de sommet non exploré voisin de 4. La deque `D` vaut alors : [3]. La liste `VISITED` vaut : [0, 1, 2, 4, 3].
- On supprime l'élément à l'extrémité gauche de `D`. Pas de sommet non exploré voisin de 3. La deque `D` vaut alors : [ ]. La liste `VISITED` vaut : [0, 1, 2, 4, 3].

La boucle `while` est terminée lorsque `D` est vide. Avec la fonction `parcourslargeur_mat`, on explore les sommets suivants : `VISITED` = [0, 1, 2, 4, 3]. On a bien réalisé un parcours en largeur.

4. Soit *n* le nombre de sommets.

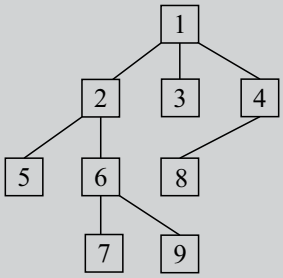
À chaque passage dans la boucle `while len(D)!=0`, on supprime un sommet à l'extrémité gauche de la deque `D`. Cette boucle sera donc exécutée au plus *n* fois.

À chaque itération de la boucle `while`, la boucle `for j in range` est exécuté au maximum *n* fois.

La complexité dans le pire des cas pour un graphe représenté par une matrice d'adjacence est quadratique en $O(n^2)$.

Exercice 12.7 : Parcours en profondeur d'un graphe

On modélise un site web par une page d'accueil qui contient des liens hypertextes permettant d'accéder à d'autres pages du site qui peuvent contenir également des liens hypertextes. Le site web contient 9 pages numérotées de 1 à 9. On considère le graphe non orienté $G=(S,A)$ représentant la structure du site web.



On considère une pile `P` et une liste `PARCOURS` qui contient la liste des sommets explorés. On pose `début=1` le sommet de départ.

Les étapes de l'algorithme de parcours en profondeur sont les suivantes :

Initialisation de l'algorithme :

La pile `P` contient le sommet de départ : `P=[début]`.

La liste `PARCOURS` contient le sommet de départ : `PARCOURS=[début]`.

Boucle tant que la pile `P` n'est pas vide :

- 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 utilisera un flag `trouve` (de valeur `True` ou `False`) qui permet de savoir si le sommet *x* possède un voisin non exploré.

1. Définir un dictionnaire `dico` représentant le graphe *G*. Chaque clé est associée à un sommet. La valeur de la clé représente la liste des sommets adjacents.

2. Écrire une fonction itérative `parcoursprofondeur` qui admet comme arguments un dictionnaire `dico` et un sommet de départ `début`. La fonction retourne la liste `PARCOURS`.

3. Que retourne `parcoursprofondeur(dico, 1)` ?

Analyse du problème

L'algorithme de parcours en profondeur (ou DFS, Depth First Search, en anglais) explore une branche en profondeur depuis un sommet avant de passer à la suivante.

On va le plus profond possible pour chaque branche. On utilise le principe LIFO (Last In, First Out : « dernier entré, premier sorti »).

1.

```
dico={1:[2, 3, 4], 2:[1, 5, 6], 3:[1], 4:[1, 8],\
  5:[2], 6:[2, 7, 9], 7:[6], 8:[4], 9:[6]}
```

Le sommet 2 est relié aux sommets 1, 5 et 6. La valeur de la clé 2 est la liste des sommets adjacents [1, 5, 6].

2. Comparaison entre les algorithmes de parcours en largeur et en profondeur :

- Parcours en largeur (BFS, Breadth First Search, en anglais) : voir exercice 12.5 « Parcours en largeur d'un graphe avec une deque ». On utilise le principe FIFO (First In, First Out : « premier entré, premier sorti »). **Tous les sommets non explorés adjacents** au sommet traité sont ajoutés dans la liste `VISITED`.
- Parcours en profondeur ou DFS : on utilise le principe LIFO (Last In, First Out : « dernier entré, premier sorti »). On parcourt tous les sommets adjacents au sommet traité avec une boucle `while i<len(L) and trouve==False`.

  Dès qu'on trouve un sommet non exploré, le flag `trouve` passe à `True`, on ajoute ce sommet dans la liste `PARCOURS` et dans la pile `P`, on quitte la boucle `while i<len(L) and trouve==False`. On ne dépile pas ce sommet ajouté et on continue l'exploration en profondeur en repartant au début de la boucle `while len(P)!=0`.

Remarque : On empile dans `P` uniquement un voisin non exploré dans le parcours en profondeur alors qu'on ajoute dans `D` tous les voisins non explorés dans le parcours en largeur.

```
def parcoursprofondeur(dico, début):
    # fonction permettant un parcours en profondeur du
    # dictionnaire dico en partant de début et retournant
    # PARCOURS la liste des sommets visités
    P=[début]
    PARCOURS=[début]
    while len(P)!=0:
        x=P.pop()     # dépile pour récupérer l'élément 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:  # cherche dans L un voisin
                                      # non exploré
```

```
                trouve=True    # on a trouvé un voisin non exploré
                P.append(L[i]) # ajoute ce sommet en haut de
                               # la pile
                PARCOURS.append(L[i]) # marque ce voisin exploré
            i+=1
        if trouve==False:
            P.pop()  # dépile le haut de la pile
    return PARCOURS

print("Parcours en profondeur d'un graphe")
PARCOURS=parcoursprofondeur(dico, 1)
print(PARCOURS)
```

On utilise très souvent trois opérations de base avec les piles : « empiler », « dépiler » et « tester si la pile est vide ». Voir exercice 11.3 « Opérations de base sur les piles » dans le chapitre « Dictionnaire, pile, file, deque ». 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]     # x est l'élément en haut de la pile
```

Remarque : Au début de la boucle `while len(P)!=0`, il faut récupérer le numéro du sommet en haut de la pile. Lorsqu'on utilise des piles, on ne peut utiliser que « empiler » et « dépiler » pour récupérer le numéro du sommet en haut de la pile. Il ne faut surtout pas dépiler ce sommet à cette étape du programme puisqu'il peut rester des sommets non explorés adjacents à ce sommet.

`PARCOURS.append(L[i])` permet de marquer le sommet que l'on a découvert lors de l'exploration.

3.

Initialisation de l'algorithme :

- La pile `P` contient le sommet de départ : `P` = [1].
- La liste `PARCOURS` contient le sommet de départ : `PARCOURS` = [1].

Étapes de la boucle `while` :

- Le haut de la pile `P` (sommet 1) possède un voisin non exploré : 2 car il n'est pas dans la liste `PARCOURS`. On ajoute alors 2 dans la liste `PARCOURS` et dans la pile `P`. On obtient : `P` = [1, 2] ; `PARCOURS` = [1, 2].
- Le haut de la pile `P` (sommet 2) possède un voisin non exploré : 5 car il n'est pas dans la liste `PARCOURS`. On ajoute alors 5 dans la liste `PARCOURS` et dans la pile `P`. On obtient : `P` = [1, 2, 5] ; `PARCOURS` = [1, 2, 5].

- Le haut de la pile `P` (sommet 5) n'a pas de voisin non exploré. On dépile 5 de la pile `P`. On obtient : `P` = [1, 2] ; `PARCOURS` = [1, 2, 5]. On remonte au sommet précédent et on continue l'exploration du sommet 2. On réalise un parcours en profondeur puisqu'on va le plus loin possible. On est bloqué au sommet 5. On remonte en arrière et on explore le sommet 2 en explorant en profondeur ce sommet.
- Le haut de la pile (sommet 2) possède un voisin non exploré : 6. On ajoute alors 6 dans la liste `PARCOURS` et dans la pile `P`. On obtient : `P` = [1, 2, 6] ; `PARCOURS` = [1, 2, 5, 6].
- Le haut de la pile (sommet 6) possède un voisin non exploré : 7. On ajoute alors 7 dans la liste `PARCOURS` et dans la pile `P`. On obtient : `P` = [1, 2, 6, 7] ; `PARCOURS` = [1, 2, 5, 6, 7].
- Le haut de la pile `P` (sommet 7) n'a pas de voisin non exploré. On dépile 7 de la pile `P`. `P` = [1, 2, 6] ; `PARCOURS` = [1, 2, 5, 6, 7]. On remonte au sommet précédent et on continue l'exploration du sommet 6.
- Le haut de la pile (sommet 6) possède un voisin non exploré : 9. On ajoute alors 9 dans la liste `PARCOURS` et dans la pile `P`. On obtient : `P` = [1, 2, 6, 9] ; `PARCOURS` = [1, 2, 5, 6, 7, 9].
- Le haut de la pile `P` (sommet 9) n'a pas de voisin non exploré. On dépile 9 de la pile `P`. On obtient : `P` = [1, 2, 6] ; `PARCOURS` = [1, 2, 5, 6, 7, 9].
- On remonte au sommet précédent et on continue l'exploration du sommet 6.
- Le sommet 6 n'a plus de sommet non exploré. On remonte au sommet 2 qui n'a plus de sommet non exploré. On remonte au sommet 1. On obtient : `P` = [1] ; `PARCOURS` = [1, 2, 5, 6, 7, 9].
- Le haut de la pile (sommet 1) possède un voisin non exploré : 3. On ajoute alors 3 dans la liste `PARCOURS` et dans la pile `P`. On obtient : `P` = [1, 3] ; `PARCOURS` = [1, 2, 5, 6, 7, 9, 3].
- Le haut de la pile `P` (sommet 3) n'a pas de voisin non exploré. On dépile 3 de la pile `P`. On obtient : `P` = [1] ; `PARCOURS` = [1, 2, 5, 6, 7, 9, 3]. On remonte au sommet précédent et on continue l'exploration du sommet 1.
- Le haut de la pile (sommet 1) possède un voisin non exploré : 4. On ajoute alors 4 dans la liste `PARCOURS` et dans la pile `P`. On obtient : `P` = [1, 4] ; `PARCOURS` = [1, 2, 5, 6, 7, 9, 3, 4].
- Le haut de la pile (sommet 4) possède un voisin non exploré : 8. On ajoute alors 8 dans la liste `PARCOURS` et dans la pile `P`. On obtient : `P` = [1, 4, 8] ; `PARCOURS` = [1, 2, 5, 6, 7, 9, 3, 4, 8].
- Le haut de la pile `P` (sommet 8) n'a pas de voisin non exploré. On dépile 8 de la pile `P`. On obtient : `P` = [1, 4] ; `PARCOURS` = [1, 2, 5, 6, 7, 9, 3, 4, 8]. On remonte au sommet précédent et on continue l'exploration du sommet 4.

- Le haut de la pile `P` (sommet 4) n'a pas de voisin non exploré. On dépile 4 de la pile `P`. On obtient : `P` = [1] ; `PARCOURS` = [1, 2, 5, 6, 7, 9, 3, 4, 8]. On remonte au sommet précédent et on continue l'exploration du sommet 1.
- Le haut de la pile `P` (sommet 1) n'a pas de voisin non exploré. On dépile 1 de la pile `P`. On obtient : `P` = [ ] ; `PARCOURS` = [1, 2, 5, 6, 7, 9, 3, 4, 8].

L'algorithme est terminé puisque la pile `P` est vide.

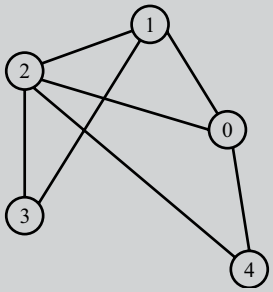
Remarques :

Dans l'algorithme du parcours en profondeur, on peut choisir d'autres parcours d'exploration :

- Le sommet 1 a trois sommets adjacents.
- On a commencé par explorer en profondeur 2 et 5 mais on aurait pu explorer un autre parcours, par exemple 4 et 8.

Exercice 12.8 : Test de connexité d'un graphe – Parcours en profondeur – Arbre couvrant

On considère le graphe non orienté $G = (S, A)$:



On utilise les listes de listes pour représenter les matrices dans Python.

1. Construire la matrice d'adjacence $\mathbf{M}_{i,j}$ du graphe G. Si deux sommets différents i et j sont reliés par une arête, alors $\mathbf{M}[i][j] = 1$ sinon $\mathbf{M}[i][j] = 0$.

2. Écrire une fonction `testgrapheconnexe_profondeur` qui admet comme argument une matrice d'adjacence `M` et retourne `True` si le graphe est connexe et `False` sinon en utilisant le **parcours en profondeur**.

3. Écrire une fonction `arbrecouvrant_profondeur` qui admet comme arguments une matrice d'adjacence `M` et un sommet de départ `début` permettant de récupérer une matrice d'adjacence représentant l'arbre couvrant correspondant au parcours en profondeur pour un **graphe non orienté et connexe**.

On considère deux listes (`PERE`, `PARCOURS`) et une pile `P`. La liste `PARCOURS` contient la liste des sommets explorés. `PERE[k]` représente le père de `PARCOURS[k]`.

Les étapes de l'algorithme sont les suivantes :

Initialisation de l'algorithme :

- La pile `P` contient le sommet de départ.
- La liste `PARCOURS` contient le sommet de départ.
- La liste `PERE` contient −1.

Boucle tant que la pile `P` n'est pas vide :

- 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.

4. Représenter l'arbre couvrant du graphe *G*.

Analyse du problème

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.

On utilise le parcours en profondeur pour construire un arbre inclus dans le graphe *G* et qui relie tous les sommets de ce graphe (voir exercice précédent « Parcours en profondeur d'un graphe »).

Pour construire l'arbre couvrant, on définit deux listes : `PARCOURS` et `PERE`. La liste `PERE` contient la liste des pères pour chaque sommet de `PARCOURS`. Il faut en effet connaître la liste des sommets parcourus et savoir comment on a atteint ce sommet.

1. La matrice d'adjacence est : $\mathbf{M} = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{pmatrix}$.

```
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 : $\mathbf{M}[i]\ [i] = 0$.

Les éléments sont symétriques par rapport à la diagonale puisque le graphe n'est pas orienté : $\mathbf{M}[i][j] = \mathbf{M}[j][i]$.

On peut déduire de la deuxième ligne de la matrice que le sommet 1 est relié aux sommets : 0, 2 et 3.

Pour la troisième ligne, le sommet 2 est relié aux sommets : 0, 1, 3 et 4.

2. Pour savoir si le graphe *G* est connexe, on récupère la liste des sommets explorés avec l'algorithme de parcours en profondeur. Si tous les sommets du graphe *G* sont dans cette liste, alors *G* est connexe.

On appelle `début` le sommet de départ. On pose par exemple `début = 0`. On aurait pu prendre un autre sommet de *G*.

```
def testgrapheconnexe_profondeur(M):
    # la fonction retourne True si le graphe est connexe
    # et False sinon pour la matrice d'adjacence M
    n=len(M)
    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
```

Dans la fonction `testgrapheconnexe_profondeur`, la liste `PARCOURS` donne la liste de tous les sommets explorés avec l'algorithme en profondeur.

`n=len(M)` retourne le nombre de lignes de la matrice `M`.

Il suffit de tester si la longueur de la liste `PARCOURS` est égale au nombre de lignes de la matrice `M` pour savoir si on a bien exploré tous les sommets du graphe.

3. L'arbre couvrant d'un graphe non orienté et connexe est un arbre inclus dans le graphe et qui relie tous les sommets du graphe *G*.

Pour construire l'arbre couvrant, il suffit de parcourir les listes `PARCOURS` et `PERE` en commençant à `PARCOURS[1]`. Le père de `PARCOURS[k]`

est `PERE[k]`. On peut ainsi remplir la matrice d'adjacence de l'arbre couvrant.

```
def arbrecouvrant_profondeur(M, début):
    # la fonction retourne l'arbre couvrant de la matrice M
    # en partant du sommet début (entier)
    # le graphe est non orienté et connexe
    n=len(M)
    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 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.
        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 :

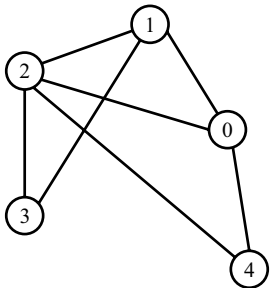
On utilise très souvent trois opérations de base avec les piles : « empiler », « dépiler » et « tester si la pile est vide ». Voir exercice 11.5 « Opérations de base sur les piles » dans le chapitre « Dictionnaire, pile, file, deque ». Les lignes suivantes permettent de récupérer le sommet *x* du haut de la pile.

```
x=P.pop()   # on dépile pour récupérer l'élément en haut de la pile
P.append(x) # on rempile x
```

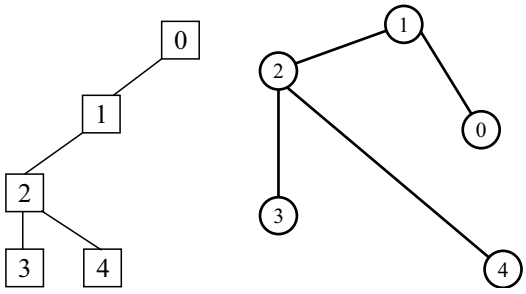
4. La liste `PARCOURS` vaut : [0, 1, 2, 3, 4].
La liste `PERE` vaut : [-1, 0, 1, 2, 2].
On obtient la matrice d'adjacence représentant l'arbre couvrant :

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}.$$

Le graphe de départ est :



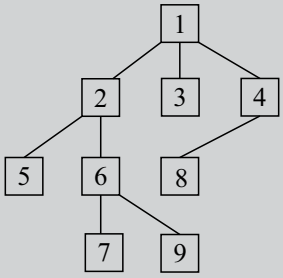
On obtient l'arbre couvrant :



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.

Exercice 12.9 : Algorithme récursif du parcours en largeur

On modélise un site web par une page d'accueil qui contient des liens hypertextes permettant d'accéder à d'autres pages du site qui peuvent contenir également des liens hypertextes. Le site web contient 9 pages numérotées de 1 à 9. On considère le graphe non orienté $G = (S, A)$ représentant la structure du site web.



La liste `PARCOURS` contient la liste des sommets parcourus par l'algorithme. On ajoute le sommet de départ `début` dans la liste `PARCOURS` initialement vide et dans la deque `D` initialement vide.

Principe de l'algorithme récursif :

- On supprime le sommet *x* à l'extrémité gauche de `D`. On ajoute tous les sommets non explorés adjacents à *x*, dans la liste `PARCOURS` et à l'extrémité droite de `D`.
- On appelle la fonction récursive avec la deque `D` et la liste `PARCOURS`.

1. Définir un dictionnaire `dico` représentant le graphe *G*. Chaque clé est associée à un sommet. La valeur de la clé représente la liste des sommets adjacents.

2. Écrire une fonction récursive `BFS_rec` qui admet comme arguments un dictionnaire `dico`, une deque `D` et une liste `PARCOURS`.

3. Le sommet de départ vaut 1. Que vaut `PARCOURS` après l'appel de la fonction `BFS_rec(dico, D, PARCOURS)` ?

Analyse du problème

L'algorithme de parcours en largeur (ou BFS, Breadth First Search, en anglais) permet de traiter les sommets adjacents à un sommet donné pour ensuite les explorer un par un. L'implémentation repose sur une deque `D` dans laquelle on place le premier sommet à l'extrémité droite de `D` initialement vide et les sommets adjacents non explorés à l'extrémité droite de `D`. On utilise le principe FIFO (First In, First Out : « premier entré, premier sorti »).

1.

```
dico={1:[2, 3, 4], 2:[1, 5, 6], 3:[1, 6], 4:[1, 6, 8],\
  5:[2], 6:[2, 7, 9], 7:[6], 8:[4],9:[6]}
```

Le sommet 2 est relié aux sommets 1, 5 et 6. La valeur de la clé 2 est la liste des sommets adjacents [1, 5, 6].

2.

- La condition d'arrêt de la fonction récursive est que la deque `D` est vide.
- On supprime le sommet *x* à l'extrémité gauche de `D` et `L=dico[x]` contient tous les sommets adjacents à *x*. La boucle `for` permet de parcourir tous les sommets adjacents à *x*. Tous les sommets adjacents non explorés sont ajoutés dans `PARCOURS` et à l'extrémité droite de `D`. Si *x* = 1, on ajoute les sommets 2, 3 et 4. On a bien un parcours en largeur.
- On appelle à nouveau la fonction récursive avec la deque `D` et la liste `PARCOURS` modifiées précédemment.

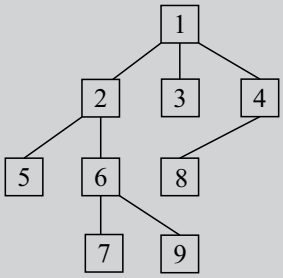
Les listes et les deques sont passées par référence et non par valeur dans les fonctions. On considère donc la même liste `PARCOURS` et la même deque `D` lors des différents appels récursifs.

```
def BFS_rec(dico, D, PARCOURS):
    # la fonction permet un parcours en largeur du dictionnaire
    # dico en partant de début et permettant d'obtenir
    # PARCOURS la liste des sommets visités
    if len(D)==0:
        return ()  # condition d'arrêt
    else:
        x=D.popleft()  # supprime le sommet à l'extrémité
                       # gauche de D
        L=dico[x]      # liste contenant les sommets adjacents à x
        for elt in L:  # parcourt les éléments de L
            if elt not in PARCOURS:
                D.append(elt)      # ajoute elt à l'extrémité
                                   # droite de D
                PARCOURS.append(elt)
        BFS_rec(dico, D, PARCOURS) # appel récursif

from collections import deque # module permettant d'utiliser
                              # les deques
D=deque()    # deque vide
début=1
D.append(début)
PARCOURS=[début]
BFS_rec(dico, D, PARCOURS)
print('Algorithme parcours en largeur récursif - Parcours :',\
      PARCOURS)
```

3. On obtient : `PARCOURS = [1, 2, 3, 4, 5, 6, 8, 7, 9]`.

Exercice 12.10 : Algorithme récursif du parcours en profondeur

On modélise un site web par une page d'accueil qui contient des liens hypertextes permettant d'accéder à d'autres pages du site qui peuvent contenir également des liens hypertextes. Le site web contient 9 pages numérotées de 1 à 9. On considère le graphe non orienté $G = (S, A)$ représentant la structure du site web.



On définit la liste `PARCOURS` qui contient la liste des sommets parcourus par l'algorithme. La liste `PARCOURS` est initialement vide.

Principe de l'algorithme récursif :

- On ajoute le sommet `s` dans la liste `PARCOURS`.
- On appelle la fonction récursive pour le premier sommet non exploré adjacent à `s`.

1. Définir un dictionnaire `dico` représentant le graphe *G*. Chaque clé est associée à un sommet. La valeur de la clé représente la liste des sommets adjacents.

2. Écrire une fonction récursive `profondeur_rec` qui admet comme arguments un dictionnaire `dico`, un sommet de départ `s` et une liste `PARCOURS`.

3. Qu'affiche le programme suivant ?

```
profondeur_rec(dico, 1, [])
print(PARCOURS)
```

Représenter l'arbre des appels de la fonction récursive `profondeur_rec (dico, 1)`.

Analyse du problème

L'algorithme de parcours en profondeur (ou DFS, Depth First Search, en anglais) explore une branche en profondeur depuis un sommet avant de passer à la suivante. On va le plus profond possible pour chaque branche.

1.

```
dico={1:[2, 3, 4], 2:[1, 5, 6], 3:[1, 6], 4:[1, 6, 8],\
  5:[2], 6:[2, 7, 9], 7:[6], 8:[4],9:[6]}
```

Le sommet 2 est relié aux sommets 1, 5 et 6. La valeur de la clé 2 est la liste des sommets adjacents [1, 5, 6].

2.

```
def profondeur_rec(dico, s, PARCOURS):
    # la fonction permet un parcours en profondeur du dictionnaire
    # dico en partant de s et permettant d'obtenir
    # PARCOURS la liste des sommets visités
    PARCOURS.append(s) # ajoute le sommet s dans liste PARCOURS
    L=dico[s]          # liste des sommets adjacents
    for elt in L:
        if elt not in PARCOURS:
            profondeur_rec(dico, elt, PARCOURS) # appel récursif
    # condition d'arrêt si tous les éléments de L ont été explorés
    # On n'a pas besoin de return car on ajoute
    # les sommets explorés au fur et à mesure dans PARCOURS
```

3. On considère le programme principal suivant :

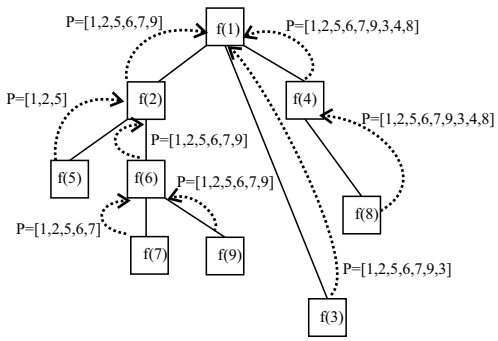
```
profondeur_rec(dico, 1, [])
print(PARCOURS)
```

Le programme Python affiche : `[1, 2, 5, 6, 7, 9, 3, 4, 8]`.

La fonction récursive est notée `f`. Pour une meilleure lisibilité, on écrit `f(1)` au lieu de `profondeur_rec(dico, 1, PARCOURS)`. La liste `PARCOURS` est notée `P`.

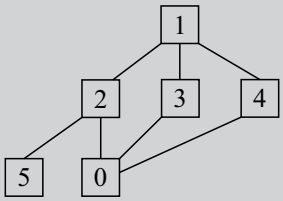
- On appelle `f(1)`. Le sommet 1 est en cours d'exploration. On l'ajoute dans la liste `P`. La liste `L` contient les sommets adjacents à 1 : `L` = [2, 3, 4]. On considère le premier sommet de la liste. Le sommet 2 n'a pas encore été exploré. On a un appel récursif `f(2)`.
- Le sommet 2 est en cours d'exploration. On l'ajoute dans la liste `P`. La liste L contient les sommets adjacents à 2 : `L` = [1, 5, 6]. On considère le premier sommet de la liste non exploré. Le sommet 5 n'a pas encore été exploré. On a un appel récursif `f(5)`.
- Le sommet 5 est en cours d'exploration. On l'ajoute dans la liste `P`. La liste L contient les sommets adjacents à 5 : `L` = [2]. Aucun élément de `L` est non exploré. On est dans la condition d'arrêt de la fonction récursive. Phase de remontée avec `P` = [1, 2, 5].
- On revient au sommet 2. On explore le sommet 6 puis appel récursif pour le sommet 7. Phase de remontée avec `P` = [1, 2, 5, 6, 7].

L'arbre ci-contre représente les différents appels de la fonction `profondeur_rec` que l'on note `f`.



Exercice 12.11 : 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 gris. Après avoir traité en largeur tous les sommets adjacents au sommet *i*, le sommet *i* est noir.

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.

On utilise les listes de listes pour représenter les matrices dans Python.

1. Construire la matrice d'adjacence $\mathbf{M}_{i,j}$ du graphe *G*. Si deux sommets différents *i* et *j* sont reliés par une arête, alors $\mathbf{M}[i][j] = 1$, sinon $\mathbf{M}[i][j] = 0$.

2. Écrire une fonction `cycle_som` qui admet comme arguments une matrice d'adjacence `M` et un sommet `début`. Cette fonction parcourt en largeur le graphe *G*. La fonction retourne `True` lorsqu'un sommet *i* adjacent à un sommet *x* n'est pas blanc et que le père de *x* n'est pas *i*. La fonction retourne `False` sinon.

3. Écrire le programme principal permettant d'afficher si un graphe connexe et non orienté possède au moins un cycle. On pourra appeler la fonction `cycle_som` à un sommet quelconque du graphe.

4. Écrire le programme principal permettant d'afficher si un graphe non orienté possède au moins un cycle. On pourra appeler la fonction `cycle_som` à chaque sommet du graphe.

Analyse du problème

On utilise le parcours en largeur du graphe.

Un chemin est simple si toutes les arêtes du chemin sont différentes. 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.

1. La matrice d'adjacence est : $\mathbf{M} = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$.

```
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 : $\mathbf{M}[i][i] = 0$.

Les éléments sont symétriques par rapport à la diagonale puisque le graphe n'est pas orienté : $\mathbf{M}[i][j] = \mathbf{M}[j][i]$.

2. Il y a des différences dans la recherche de cycle pour les graphes orientés et pour les graphes non orientés :

- Pour un graphe orienté, 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 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" 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
```

3. On considère un sommet quelconque pour le graphe connexe et non orienté.

```
début=0
if cycle_som(M, début)==True:
    print('Le graphe connexe et non orienté possède au moins'\
          ' un cycle.')
else:
    print('Le graphe connexe et non orienté ne possède pas de'\
          ' cycle en partant du sommet ',début)
```

4. On applique la fonction `cycle_som` à chaque sommet du graphe.

```
def rec_cycle_larg(M):
    # la fonction retourne False si la matrice M ne possède
    # pas de cycle
    n=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 trouvé
    return False

if rec_cycle_larg(M)==True:
    print('Le graphe possède au moins un cycle.')
else:
    print('Le graphe ne possède pas de cycle.')
```

Partie 10

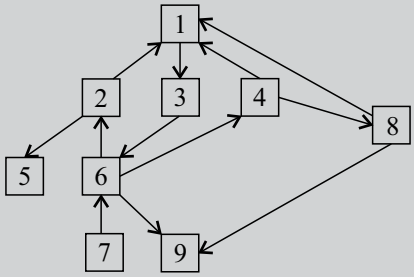
Recherche d'un plus court chemin

Plan

13 Recherche d'un plus court chemin (sauf TSI et TPC)

Exercice 13.1 : Recherche d'un plus court chemin, graphe orienté

On considère le graphe orienté $G = (S, A)$:



`L.reverse()` permet d'inverser les éléments de la liste `L`.

On considère dans l'algorithme une liste `PARCOURS` et un dictionnaire `PERE` :

- La liste `PARCOURS` contient la liste des sommets d'un plus court chemin du sommet de départ `début` jusqu'au sommet d'arrivée `fin` différent de `début`.
- `PERE[i]` représente le père du sommet *i* lors du parcours en largeur du graphe depuis le sommet de départ `début`.

1. Définir un dictionnaire `dico` représentant le graphe *G*. La clé associée à chaque sommet représente la liste des successeurs.

2. Écrire une fonction itérative `BFS` qui admet comme arguments un dictionnaire `dico` et un sommet de départ `début`. La fonction retourne un dictionnaire `PERE` en utilisant l'algorithme de parcours en largeur.

3. La liste `PARCOURS` est initialement vide.

Principe de l'algorithme d'un plus court chemin :

Pour obtenir les sommets d'un plus court chemin de `début` jusqu'à `fin`, il faut remonter dans l'arborescence du dictionnaire `PERE` depuis le sommet `fin` jusqu'à la racine `début`.

Écrire une fonction récursive `pluscourtchemin` qui admet comme arguments un sommet de départ `début`, un sommet d'arrivée `fin`, un dictionnaire `PERE` et une liste `PARCOURS` permettant d'obtenir un plus court chemin de `début` jusqu'à `fin`.

4. Écrire le programme principal permettant d'afficher un plus court chemin entre le sommet de départ `début` et le sommet d'arrivée `fin`. Qu'obtient-on pour `début = 1` et `fin = 8` ?

Analyse du problème

Un plus court chemin de `début` jusqu'à `fin` est le chemin comportant le moins d'arcs. On utilise l'algorithme de parcours en largeur (BFS, Breadth First Search, en anglais) permettant de traiter les sommets adjacents à un sommet donné pour ensuite les explorer un par un. Voir exercice 12.5 « Parcours en largeur d'un graphe avec une deque » dans le chapitre « Graphes ».

1.

```
dico={1:[3], 2:[1, 5], 3:[6],\
      4:[1, 8], 5:[], 6:[2, 4, 9],\
      7:[6], 8:[1, 9], 9:[]}
```

Le sommet 2 a deux successeurs : 1 et 5. La valeur de la clé 2 est la liste des successeurs [1, 5]. Le sommet 6 n'est pas le successeur du sommet 4.

2. Les sommets déjà visités sont marqués pour éviter d'explorer plusieurs fois un même sommet. La liste `VISITED` contient les sommets visités.

Les étapes de l'algorithme de parcours en largeur sont les suivantes :

Initialisation de l'algorithme :

- Mettre le sommet de départ dans la deque `D` initialement vide.
- La liste `VISITED` contient le sommet de départ : `VISITED = [début]`.

Boucle tant que la deque `D` n'est pas vide :

- Supprimer le sommet *x* à l'extrémité gauche de `D`.
- Ajouter dans `VISITED` et à l'extrémité droite de `D` les sommets non explorés adjacents au sommet *x*.

```
def BFS(dico, début):
    # la fonction renvoie le dictionnaire PERE avec un parcours
    # en largeur pour le dictionnaire dico
    from collections import deque   # module permettant d'utiliser
                                    # les deques
    D=deque()          # deque vide
    D.append(début)    # ajoute le sommet de départ
    PERE={ }           # dictionnaire vide
    VISITED=[début]
    while len(D)!=0:
        x=D.popleft() # supprime le sommet x à l'extrémité
                      # gauche de D
        L=dico[x]     # liste contenant les sommets adjacents à x
        for elt in L: # parcourt les éléments de L
            if elt not in VISITED:  # teste si le sommet n'a pas
                                    # déjà été exploré
                D.append(elt) # ajoute le sommet elt à l'extrémité
                              # droite de D
                VISITED.append(elt) # le sommet elt a été exploré
                PERE[elt]=x # ajoute clé, valeur dans le dico PERE
    return PERE
```

La ligne `PERE[elt]=x` permet d'ajouter `elt` (clé) et `x` (valeur de la clé) dans le dictionnaire `dico`.

3.

```
def pluscourtchemin(début, fin, PERE, PARCOURS):
    # la fonction permet d'avoir un plus court chemin
    # de début (int) à fin (int) dans la liste PARCOURS
    # à partir du dictionnaire PERE
    if début==fin:
        PARCOURS.append(fin)
        return()              # condition d'arrêt
    elif PERE[fin]=='':
        PARCOURS=[]
        return ()             # condition d'arrêt
    else:
        PARCOURS.append(fin)  # on ajoute le sommet fin dans
                              # la liste PARCOURS
        pluscourtchemin(début, PERE[fin], PERE, PARCOURS)
                              # appel récursif
```

4.

```
début, fin=1, 8
PERE=BFS(dico, début)
if fin not in PERE:     # teste si le sommet fin est dans
                        # le dictionnaire PERE
    print("Il n'y a pas de chemin entre", début, "et", fin,".")
else:
    PARCOURS=[]
    pluscourtchemin(début, fin, PERE, PARCOURS)
    PARCOURS.reverse()  # inverse les éléments de la liste
    print('Un plus court chemin :', PARCOURS)
```

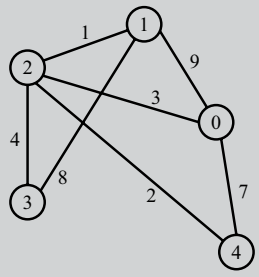
Si le sommet `fin` n'est pas dans le dictionnaire `PERE`, alors il n'y a pas de chemin entre `début` et `fin`.

Le programme Python affiche :

```
Parcours : [1, 3, 6, 4, 8]
```

Exercice 13.2 : Algorithme de Dijkstra

On considère le graphe non orienté $G = (S, A)$, où le nombre situé sur l'arête joignant deux villes (ou sommets) est leur distance :



On utilise les listes de listes pour représenter les matrices dans Python.

`L.reverse()` permet d'inverser les éléments de la liste `L`.

1. Construire la matrice d'adjacence $(\mathbf{M}_{i,j})_{0\leq i,j\leq n-1}$ (appelée également **matrice de distance)** du graphe *G*, définie par :

Pour tous les indices *i, j*, $\mathbf{M}_{i,j}$ représente la distance entre les villes d'origine *i* et d'extrémité *j*.

Lorsque les villes ne sont pas reliées, cette distance vaut l'infini. On définit la variable `inf=1e10` qui représente une distance infinie.

2. 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 la liste `VILLES` contenant les informations suivantes pour chaque ville : [ville précédente sur le chemin, distance parcourue depuis la ville de départ, ville sélectionnée ou non (booléen vrai ou faux)].

a) Initialisation de l'algorithme :

Toutes les villes sont non sélectionnées sauf la ville de départ. Pour cette ville, la distance parcourue vaut 0.

Pour les villes non sélectionnées, les distances parcourues depuis la ville de départ sont initialisées à l'infini.

On définit une variable `position` qui correspond à la ville pour laquelle l'algorithme est appliqué. Cette variable prend initialement la valeur `départ`.

b) Tant que la variable `position` n'est pas égale à la variable `arrivée`, répéter les opérations suivantes pour toutes les villes *i* non sélectionnées :

- Calculer la variable `somme` = distance entre la ville `départ` et la ville `position` + distance entre la ville `position` et la ville *i*.
- Si la variable `somme` est inférieure à la distance entre la ville `départ` et la ville *i*, alors remplacer cette distance par `somme`. La ville précédente sur le chemin est alors `position`.

Chercher la ville (parmi les villes non sélectionnées) pour laquelle la distance entre celle-ci et la ville `départ` est la plus petite. Cette ville définit alors la nouvelle valeur de la variable `position` et cette ville devient sélectionnée.

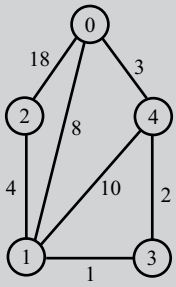
Écrire une fonction `dijkstra` qui admet comme arguments d'entrée la matrice d'adjacence `M`, la ville `départ` et la ville `arrivée`. Cette fonction retourne le chemin suivi ainsi que la distance parcourue entre `départ` et `arrivée` en utilisant l'algorithme de Dijkstra.

Écrire le programme principal permettant de déterminer un plus court chemin entre la ville de départ 3 et la ville d'arrivée 0. Le programme affichera le chemin suivi ainsi que la distance parcourue.

3. Pourquoi cet algorithme est appelé algorithme glouton ?

4. Calculer la complexité de cet algorithme dans le pire des cas.

5. On considère le graphe suivant :



Écrire le programme permettant de déterminer un plus court chemin entre la ville de départ 0 et la ville d'arrivée 2. Le programme affichera le chemin suivi ainsi que la distance parcourue.

Analyse du problème

On étudie l'algorithme de Dijkstra, qui un algorithme de plus court chemin. On définit la matrice d'adjacence qui contient l'ensemble des distances entres les villes. La longueur d'un chemin est la somme des poids des arêtes qui le constituent.

1. La matrice d'adjacence est : $\mathbf{M} = \begin{pmatrix} 0 & 9 & 3 & \infty & 7 \\ 9 & 0 & 1 & 8 & \infty \\ 3 & 1 & 0 & 4 & 2 \\ \infty & 8 & 4 & 0 & \infty \\ 7 & \infty & 2 & \infty & 0 \end{pmatrix}$.

Remarque : On peut se reporter à l'exercice 12.1 « Matrice d'adjacence » dans le chapitre « Graphes » pour avoir plus d'explications sur la construction de cette matrice.

2.

- Dans l'algorithme de Dijkstra, on calcule la distance de `départ` à toutes les villes *i* non sélectionnés en passant par la ville `position`. On cherche ensuite le minimum des distances pour les villes non sélectionnées. Ce minimum permet de déclarer une ville sélectionnée. Cette distance est définitive.

  On définit un sous-graphe *G'*. Dans l'état initial, *G'* contient uniquement la ville `départ`. À chaque étape de la boucle `while(position!=arrivée)`, on ajoute la ville `position` dans *G'*. Toutes les distances des villes de *G'* sont définitives.

- Dans l'algorithme de Floyd-Warshall (voir exercice 14.4 « Algorithme de Floyd-Warshall » dans le chapitre « Programmation dynamique »), on peut trouver ultérieurement une distance encore plus petite en passant par d'autres sommets avec des arêtes de poids négatif. Il faut recalculer toutes les distances à l'étape suivante.

On définit une liste `VILLES` qui contient pour chaque ville *i* :

- `VILLES[i][0]` : ville précédente.
- `VILLES[i][1]` : $g(i)$ = distance entre la ville `départ` et la ville *i*.
- `VILLES[i][2]` : `True` si la ville *i* est sélectionnée, sinon `False`.

a) Initialisation de l'algorithme :

Toutes les villes sont non sélectionnées (`VILLES[i][2] = False`) sauf la ville de départ. Pour cette ville, la distance parcourue vaut 0. Pour les villes non sélectionnées, les distances parcourues depuis la ville de départ sont initialisées à l'infini : `VILLES[i][1] = inf`. La variable `position` prend initialement la valeur `départ`.

b) On définit une boucle `while (position!=arrivée)` tant que la variable `position` (sommet appartenant à la frontière entre les villes sélectionnées et les villes non sélectionnées) n'est pas égale à la variable `arrivée`.

- La boucle `for i in range(n)` avec le test `if VILLES [i][2]==False` permet de parcourir toutes les villes non sélectionnées. On calcule `somme` = `VILLES[position][1]` + `M[position][i]` (= distance entre `départ` et `position` + distance entre `position` et *i*). Si la nouvelle distance `somme` est inférieure à `VILLES[i][1]`, alors `VILLES[i][1] = somme` et la ville `position` est la ville précédente de *i* : `VILLES[i][0]` = `position`.
- On cherche la ville `indice` (parmi les villes non sélectionnées) pour laquelle la distance `VILLES[indice][1]` est la plus petite.
- Ce sommet `indice` définit alors la nouvelle valeur de la variable `position` et cette ville devient sélectionnée.
- Si `indice` = `position`, on ne peut pas atteindre la ville d'arrivée.

```
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)         # nb de villes = nb de lignes de M
    VILLES=init(départ, nb_villes)   # initialisation de la
                                     # liste VILLES
    # l'algorithme est appliqué pour la ville position
    position=départ
    while (position!=arrivée):
        indice=position
        for i in range(nb_villes):   # i décrit toutes les villes
            if VILLES[i][2]==False : # ville non sélectionnée
                somme=VILLES[position][1]+M[position][i]
                if somme<VILLES[i][1]:
                    VILLES[i][1]=somme
                       # nouvelle valeur de la distance à
                       # la ville de départ
                    VILLES[i][0]=position # nouvelle ville
                                     # précédente sur le chemin
        # recherche du minimum des distances pour les villes
        # non sélectionnées
        val_min=inf
        for i in range(nb_villes):
            if VILLES[i][2]==False and VILLES[i][1]<val_min:
                indice=i
                val_min=VILLES[i][1]
        if indice==position:
            return [],inf            # on n'atteint pas arrivée
        else:
            VILLES[indice][2]=True   # cette ville est sélectionnée
            position=indice  # nouvelle valeur de la variable
                             # position

    # liste des villes parcourues
    i=arrivée
    L=[arrivée]  # L = liste des villes parcourues en sens inverse
    while (i!=départ):
        i=VILLES[i][0]
        L.append(i)
    L.reverse()  # il faut inverser la liste L pour obtenir la
                 # liste des villes parcourues dans le sens direct
    return L, VILLES[arrivée][1]

# initialisation du programme
M=[[0, 9, 3, inf, 7], [9, 0, 1, 8, inf], [3, 1, 0, 4, 2],\
   [inf, 8, 4, 0, inf], [7, inf, 2, inf, 0]]
```

```
départ, arrivée=3, 0  # ville de départ et ville d'arrivée
L, dist=dijkstra(M, départ, arrivée)
print("Algorithme de Dijkstra - Chemin suivi : ", L) # liste des
                                                     # villes
print ("Distance parcourue = ", dist)
```

Le programme Python affiche :

```
Chemin suivi : [3, 2, 0]
Distance parcourue = 7
```

Remarque :

On considère l'exemple suivant pour expliquer l'algorithme de Dijkstra.

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 : `VILLES`=[[−1,inf,False],[−1, inf,False],[−1, inf,False],[−1,0,True], [−1,inf,False]]. Les valeurs −1 ne sont pas significatives puisque la distance est infinie ou la ville est `départ`.

Algorithme

- On définit la variable `position`, qui est la ville atteinte avec 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 = VILLES[position][1] + M[position][i]`) et la distance depuis la ville de départ (`VILLES[i][1]`).
- Chercher pour toutes les villes *X* précédentes celle où la variable « distance depuis la ville de départ » est minimale. Cette ville définit alors la nouvelle valeur de la variable `position` et la variable « ville sélectionnée » passe à `True`.

1re itération : position = 3

- On parcourt les villes *X* non sélectionnées : 0, 1, 2 et 4.

  On obtient alors : `VILLES`=[[−1,inf,False],[3,8,False],[3,4,False],[−1,0,True], [−1,inf,False]].
- On cherche dans les villes *X* non sélectionnées (variable `False`) celle où la variable « distance depuis la ville de départ » est minimale. C'est la ville 2.

  On obtient alors : `position = 2` et `VILLES`=[[−1,inf,False],[3,8,False], [3,4,True],[−1,0,True],[−1,inf,False]].

2e itération : position = 2

- On parcourt les villes *X* non sélectionnées : 0, 1, 4.

  On obtient alors : `VILLES`=[[2,7,False],[2,5,False],[3,4,True],[−1,0,True],[2,6,False]].
- On cherche dans les villes *X* non sélectionnées (variable `False`) celle où la variable « distance depuis la ville de départ » est minimale. C'est la ville 1.

  On obtient alors : `position` = 1 et `VILLES`=[[2,7,False],[2,5,True],[3,4,True],[−1,0,True],[2,6,False]].

3e itération : position = 1

- On parcourt les villes *X* non sélectionnées : 0, 4.

  On obtient alors : `VILLES`=[[2,7,False],[2,5,True],[3,4,True],[−1,0,True],[2,6,False]].
- On cherche dans les villes *X* non sélectionnées (variable `False`) celle où la variable « distance depuis la ville de départ » est minimale. C'est la ville 4.

  On obtient alors : `position` = 4 et `VILLES`=[[2,7,False],[2,5,True],[3,4,True],[−1,0,True],[2,6,True]].

4e itération : position = 4

- On parcourt les villes *X* non sélectionnées : 0.

  On obtient alors : `VILLES`=[[2,7,False],[2,5,True],[3,4,True],[−1,0,True],[2,6,True]].
- On cherche dans les villes *X* non sélectionnées (variable `False`) celle où la variable « distance depuis la ville de départ » est minimal. C'est la ville 0.

  On obtient alors : `position` = 0 et `VILLES` = [[2,7,True], [2,5,True], [3,4,True], [−1,0,True], [2,6,True]].

Pour obtenir le trajet, on part de la ville d'arrivée, la distance minimale entre la ville de départ et la ville d'arrivée est obtenue par `VILLES[arrivée][1]` = 7. Pour obtenir le trajet, on obtient la ville précédente avec `VILLES[arrivée][0]` = 2 et, de proche en proche, on remonte à la ville de départ.

3. L'algorithme glouton (greedy en anglais) repose sur l'utilisation de sous-problèmes. Lorsqu'on est rendu à la ville `position`, on fait un choix local qui paraît être le meilleur en choisissant, dans la liste des villes non sélectionnées, la ville suivante `indice` dont la distance entre `départ` et `indice` est la plus petite. Ce choix n'est pas remis en cause ultérieurement.

4. Soit *n* le nombre de villes.

Initialisation de la liste `VILLES` : 1 opération et, pour chaque valeur de *i* : 1 comparaison et 1 affectation. La boucle `for` est parcourue *n* fois. On a donc 2*n*+1 opérations élémentaires.

Dans le pire des cas, on a (n–1) itérations dans la boucle `while`. Pour chaque valeur de `position`, on a :

- 1 comparaison, 1 calcul de somme, 1 test et deux affectations pour chaque valeur de i : 5n opérations élémentaires.
- 1 affectation : 1 opération élémentaire.
- Recherche du minimum : trois comparaisons et deux affectations pour chaque valeur de i : 5n opérations élémentaires.
- 2 affectations : 2 opérations élémentaires.

On a donc $(2n+1)+(n-1)\times(5n+1+5n+2)=2n+1+(n-1)(10n+3)$ opérations élémentaires.

La complexité dans le pire des cas est quadratique en $O(n^2)$.

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

5. La matrice d'adjacence est : $\mathbf{M}_2 = \begin{pmatrix} 0 & 8 & 18 & \infty & 3 \\ 8 & 0 & 4 & 1 & 10 \\ 18 & 4 & 0 & \infty & \infty \\ \infty & 1 & \infty & 0 & 2 \\ 3 & 10 & \infty & 2 & 0 \end{pmatrix}$.

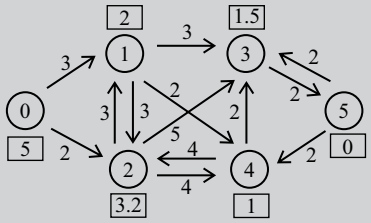
```
M2=[[0, 8, 18, inf, 3], [8, 0, 4, 1, 10], [18, 4, 0, inf, inf],\
    [inf, 1, inf, 0,2], [3, 10, inf, 2,0]]
départ2, arrivée2=0, 2       # ville de départ et ville d'arrivée
L2, dist2=dijkstra(M2, départ2, arrivée2)
print("Trajet suivi : ", L2) # affichage de la liste des villes
print ("Distance parcourue = ", dist2)
```

Le programme Python affiche :

```
Trajet suivi : [0, 4, 3, 1, 2]
Distance parcourue = 10
```

Exercice 13.3 : Algorithme A*

On considère le graphe orienté $G = (S, A)$ qui représente le réseau routier d'un département en prenant en compte le sens de la circulation. Une route à sens unique est représentée par un arc dont le poids est la distance en kilomètres entre deux sommets (ou deux villes). Les distances indiquées dans les rectangles sur le graphe ci-dessous représentent les distances à vol d'oiseau entre les sommets et le sommet d'arrivée 5.



Une heuristique est un algorithme qui calcule rapidement une solution pouvant être approximative. On utilise la distance euclidienne (ou distance à vol d'oiseau) pour estimer le coût restant $h(i)$ permettant d'atteindre le sommet `arrivée` à partir du sommet *i*.

On utilise les listes de listes pour représenter les matrices dans Python.

`L.reverse()` permet d'inverser les éléments de la liste `L`.

1. Construire la matrice d'adjacence $(\mathbf{M}_{i,j})_{0 \leq i,j \leq n-1}$ (appelée également **matrice de distance**) du graphe *G*, définie par :

Pour tous les indices *i, j*, $\mathbf{M}_{i,j}$ représente la distance entre les sommets d'origine *i* et d'extrémité *j*.

Lorsque les sommets ne sont pas reliés, cette distance vaut l'infini. On définit la variable `inf=1e10` qui représente une distance infinie.

2. L'algorithme A* est une variante de l'algorithme de Dijkstra. On dispose pour chaque sommet *i* d'une estimation du coût restant pour atteindre le sommet `arrivée` à partir du sommet *i* : $h(i)$ = distance euclidienne (ou distance à vol d'oiseau) entre le sommet *i* et le sommet `arrivée`. On définit la fonction d'évaluation *f* telle que : $f(i) = g(i) + h(i)$.

- $g(i)$ est le coût réel du chemin optimal entre le sommet `départ` et le sommet *i* dans la partie déjà explorée ;
- $h(i)$ est le coût estimé du chemin qui reste à parcourir entre *i* et `arrivée`.

On définit la liste `SOMMETS` contenant les informations suivantes pour chaque sommet :

[sommet précédent sur le chemin, distance parcourue depuis le sommet de départ,

sommet sélectionné ou non (booléen vrai ou faux), distance évaluée entre `départ` et `arrivée`]

a) Initialisation de l'algorithme A* :

Tous les sommets sont non sélectionnés sauf le sommet de départ. Pour ce sommet, la distance parcourue vaut 0. Pour les sommets non sélectionnés, les distances parcourues depuis le sommet de départ sont initialisées à l'infini. On définit une variable `position` qui correspond au sommet pour lequel l'algorithme est appliqué. Cette variable prend initialement la valeur `départ`.

b) Tant que la variable `position` n'est pas égale à la variable `arrivée` :

- Chercher le sommet `indice` (parmi les sommets non sélectionnés) pour lequel la distance *f* (`indice`) entre `départ` et `arrivée` est la plus petite.
- Ce sommet `indice` définit alors la nouvelle valeur de la variable `position` et ce sommet devient sélectionné.

On définit la liste `H=[5, 2, 3.2, 1.5, 1, 0]` telle que `H[i]` = distance euclidienne entre le sommet *i* et le sommet `arrivée` = 5.

Écrire une fonction `algoA` qui admet comme arguments d'entrée la matrice d'adjacence `M`, la liste `H`, le sommet `départ` et le sommet `arrivée`. Cette fonction retourne le chemin suivi ainsi que la distance parcourue entre `départ` et `arrivée` en utilisant l'algorithme A*.

3. Écrire le programme principal permettant de déterminer un plus court chemin entre le sommet de départ 0 et le sommet d'arrivée 5. Le programme affichera le chemin suivi ainsi que la distance parcourue.

Analyse du problème

La fonction *h* est une fonction heuristique telle que $h(i)$ est le coût estimé du chemin qui reste à parcourir entre le sommet *i* et le sommet `arrivée`. On utilise la distance à vol d'oiseau dans ce problème alors que, dans l'exercice suivant « Variantes de l'algorithme A* – Distance de Manhattan », on utilise la distance de Manhattan. L'idée est de choisir un sommet qui semble être le plus prêt du sommet d'arrivée.

La fonction d'évaluation permet de déterminer quel sommet est sélectionné en premier, c'est-à-dire retiré de la frontière entre les sommets sélectionnés et les sommets non sélectionnés : $f(i) = g(i) + h(i)$ renvoie une estimation de la distance entre le sommet de départ et le sommet d'arrivée en passant par le sommet *i* du graphe. On considère un graphe orienté dont le poids des arcs est un réel positif.

1. La matrice d'adjacence est : $\mathbf{M} = \begin{pmatrix} 0 & 3 & 2 & \infty & \infty & \infty \\ \infty & 0 & 3 & 3 & 2 & \infty \\ \infty & 3 & 0 & 5 & 4 & \infty \\ \infty & \infty & \infty & 0 & \infty & 2 \\ \infty & \infty & 4 & 2 & 0 & \infty \\ \infty & \infty & \infty & 2 & 2 & 0 \end{pmatrix}$.

Comme le graphe est orienté, la matrice n'est pas nécessairement symétrique.

2. On définit une liste `SOMMETS` qui contient pour chaque sommet *i* :

- `SOMMETS[i][0]` : sommet précédent ;
- `SOMMETS[i][1]` : $g(i)$ = distance entre le sommet `départ` et le sommet *i* ;
- `SOMMETS[i][2]` : `True` si le sommet *i* est sélectionné, sinon `False` ;
- `SOMMETS[i][3]` : $f(i)$ = distance entre le sommet `départ` et le sommet `arrivée`.

La méthode gloutonne repose sur l'utilisation de sous-problèmes. Lorsqu'on est rendu au sommet `position`, on fait un choix local qui paraît être le meilleur en choisissant, dans la liste des sommets non sélectionnés, le sommet suivant `indice` dont la distance *f*(`indice`) entre `départ` et `arrivée` est la plus petite. Ce choix n'est pas remis en cause ultérieurement. On privilégie les sommets « qui semblent » nous rapprocher de la destination.

a) Initialisation de l'algorithme :

Tous les sommets sont non sélectionnés (`SOMMETS[i][2]=False`) sauf le sommet de départ. Pour ce sommet, la distance parcourue vaut 0. Pour les sommets non sélectionnés, les distances parcourues depuis le sommet de départ sont initialisées à l'infini : `SOMMETS[i][1]=inf` et `SOMMETS[i][3]=inf`. La variable `position` prend initialement la valeur `départ`.

b) On définit une boucle `while(position!=arrivée)` : tant que la variable `position` (sommet appartenant à la frontière entre les sommets sélectionnés et les sommets non sélectionnés) n'est pas égale à la variable `arrivée`.

- La boucle `for i in range(n)` avec le test `if SOMMETS[i][2]==False` permet de parcourir tous les sommets non sélectionnés. On calcule `g_i=SOMMETS[position][1]+M[position][i]` (= distance entre `départ` et `position` + distance entre `position` et *i*) et `f_i=g_i+H[i]`.
- Si la nouvelle distance `f_i` entre `départ` et `arrivée` est inférieure à `SOMMETS[i][3]`, alors `SOMMETS [i][3]=f_i`. On met à jour également la distance entre le sommet `départ` et le sommet `i` : `SOMMETS[i][1]=g_i`. Le sommet `position` est le sommet précédent de `i` : `SOMMETS[i][0]=position`.

- On cherche le sommet `indice` (parmi les sommets non sélectionnés) pour lequel la distance f(`indice`) entre `départ` et `arrivée` est la plus petite.
- Ce sommet `indice` définit alors la nouvelle valeur de la variable `position` et ce sommet devient sélectionné.
- Si `indice` = `position`, on ne peut pas atteindre le sommet d'arrivée.

```
inf=1e10      # variable représentant l'infini

def init3(départ, n):  # n = nombre de sommets dans le graphe
    # la fonction initialise la liste SOMMETS à partir de départ
    # (int)
    SOMMETS=[]          # initialisation de la liste SOMMETS
    for i in range(n): # i varie entre 0 inclus et n exclu
        if i==départ:
            SOMMETS.append([-1, 0, True, 0])
                # la valeur -1 n'est pas utilisée
                # True : uniquement le sommet de départ est
                # sélectionné
        else:
            SOMMETS.append([-1, inf, False, inf])
                # la valeur -1 n'est pas utilisée car distance
                # infinie
                # False : sommet non sélectionné
    return SOMMETS

def algoA(M, H, départ, arrivée):
    # la fonction permet d'avoir 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
    SOMMETS=init3(départ, n) # initialisation de la liste SOMMETS
    position=départ
    while (position!=arrivée):
        indice=position
        for i in range(n):   # i décrit tous les sommets
            if SOMMETS[i][2]==False: # sommet non sélectionné
                g_i=SOMMETS[position][1]+M[position, i]
                            # g(i) = coût réel entre départ et i
                h_i=H[i]     # coût estimé entre i et arrivée
                f_i=g_i+h_i
                if f_i<SOMMETS[i][3]:
                    SOMMETS[i][1]=g_i # distance départ->i = g(i)
                    SOMMETS[i][3]=f_i
                                # distance départ->arrivée = f(i)
                    SOMMETS[i][0]=position
                                # sommet précédent sur le chemin
        # recherche du minimum des distances départ->arrivée
        # pour les sommets non sélectionnés
        val_min=inf         # initialisation de val_min à +infini
```

```
        for i in range(n):  # i varie entre 0 inclus et n exclu
            if SOMMETS[i][2]==False and SOMMETS[i][3]<val_min:
                indice=i
                val_min=SOMMETS[i][3]  # f(i)
        if indice==position:
            return [],inf   # on n'atteint pas le sommet arrivée
        else:
            SOMMETS[indice][2]=True  # ce sommet est sélectionné
            position=indice          # nouvelle valeur de position

    # liste des sommets parcourus
    i=arrivée
    L=[arrivée]  # L = liste des sommets parcourus en sens inverse
    while (i!=départ):
        i=SOMMETS[i][0]
        L.append(i)
    L.reverse()  # il faut inverser la liste L pour obtenir
                 # la liste des sommets parcourus dans le sens
                 # direct
    return L, SOMMETS[arrivée][1]
```

3.

```
inf=float("inf")
M=[[0,3,2,inf,inf,inf],[inf,0,3,3,2,inf],\
   [inf,3,0,5,4,inf],[inf,inf,inf,0,inf,2],\
   [inf,inf,4,2,0,inf],[inf,inf,inf,2,2,0]]
départ, arrivée=0, 5
H=[5, 2, 3.2, 1.5, 1, 0]
L2, dist2=algoA(M, H, départ, arrivée)
print('Algo A* chemin :',L2,'; distance',dist2)
```

Le programme Python affiche :

```
Algo A* chemin : [0, 1, 3, 5]
distance = 8
```

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. L'algorithme de Floyd-Warshall (voir exercice 14.4 « Algorithme de Floyd-Warshall » dans le chapitre « Programmation dynamique ») permet de traiter des arcs dont le poids est négatif.

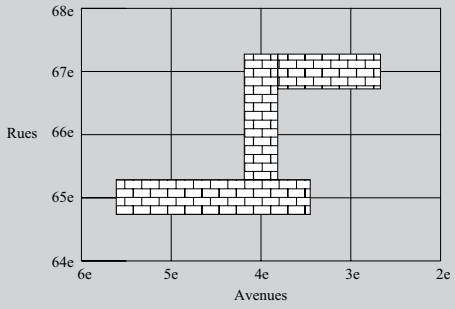
Exercice 13.4 : Variantes de l'algorithme A*, distance de Manhattan

On considère le graphe orienté $G = (S, A)$ qui représente le quartier de Manhattan. Les routes sont à double sens et certaines sont bloquées à cause de travaux (rectangles avec des briques). Les n sommets du graphe sont les intersections des routes : le sommet 11 représente l'intersection de la 5e avenue avec la 66e rue. On considère que la distance entre deux sommets adjacents vaut 1. On suppose que le nombre d'avenues est égal au nombre de rues.

Position des 25 sommets :

0	1	2	3	4
5	6	~~7~~	~~8~~	9
10	11	~~12~~	13	14
15	~~16~~	~~17~~	18	19
20	21	22	23	24

Les sommets barrés ne sont pas accessibles à cause de travaux.



Une heuristique est un algorithme qui calcule rapidement une solution pouvant être approximative. On utilise la distance de Manhattan pour estimer le coût restant ($h(i)$ = nombre d'avenues et de rues entre le sommet i et le sommet `arrivée`) permettant d'atteindre le sommet `arrivée` à partir du sommet i. On définit la variable `inf=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`.

1. Définir un dictionnaire `dico` représentant le graphe G. La clé associée à chaque sommet représente la liste des sommets adjacents.

2. Écrire une fonction `listeH` qui admet comme arguments d'entrée le sommet `arrivée` et le nombre de sommets n. Cette fonction retourne la liste $H = \left[h(0), h(1), ..., h(i), ..., h(n-1)\right]$ telle que $h(i)$ = distance de Manhattan entre le sommet i et le sommet `arrivée`.

3. L'algorithme glouton BFS (Best First Search : meilleure première recherche) permet de déterminer un plus court chemin entre le sommet `départ` et le sommet `arrivée`.

a) Initialisation de l'algorithme BFS :

Tous les sommets sont non sélectionnés sauf le sommet de départ. Pour ce sommet, la distance parcourue vaut 0. Pour les sommets non sélectionnés, les distances parcourues depuis le sommet de départ sont initialisées à l'infini. On définit une variable `position` qui correspond au sommet pour lequel l'algorithme est appliqué. Cette variable prend initialement la valeur `départ`.

b) Tant que la variable `position` n'est pas égale à la variable `arrivée` :

- Chercher le sommet `indice` (parmi les sommets non sélectionnés) pour lequel la distance $h(\text{indice})$ entre `indice` et `arrivée` est la plus petite.
- Ce sommet `indice` définit alors la nouvelle valeur de la variable `position` et ce sommet devient sélectionné.

Écrire une fonction `BFS` qui admet comme arguments d'entrée un dictionnaire `dico`, le sommet `départ` et le sommet `arrivée`. Cette fonction retourne le chemin suivi ainsi que la distance parcourue entre `départ` et `arrivée` en utilisant l'algorithme BFS.

4. On dispose pour chaque sommet i d'une estimation du coût restant pour atteindre le sommet `arrivée` à partir du sommet i : $h(i)$ = distance de Manhattan entre le sommet i et le sommet `arrivée`. On pose w un réel compris entre 0 et 1. On définit la fonction d'évaluation f telle que : $f(i) = (1-w)\cdot g(i) + w \cdot h(i)$.

- $g(i)$ est le coût réel du chemin optimal entre `départ` et i dans la partie déjà explorée ;
- $h(i)$ est le coût estimé du chemin qui reste à parcourir entre i et `arrivée`.

a) Initialisation de l'algorithme :

Tous les sommets sont non sélectionnés sauf le sommet de départ. Pour ce sommet, la distance parcourue vaut 0. Pour les sommets non sélectionnés, les distances parcourues depuis le sommet de départ sont initialisées à l'infini. On définit une variable `position` qui correspond au sommet pour lequel l'algorithme est appliqué. Cette variable prend initialement la valeur `départ`.

b) Tant que la variable `position` n'est pas égale à la variable `arrivée` :

- Chercher le sommet `indice` (parmi les sommets non sélectionnés) pour lequel la distance f (`indice`) entre `départ` et `arrivée` est la plus petite.
- Ce sommet `indice` définit alors la nouvelle valeur de la variable `position` et ce sommet devient sélectionné.

Écrire une fonction `MANHATTAN` qui admet comme arguments d'entrée un dictionnaire `dico`, le sommet `départ`, le sommet `arrivée` et le réel `w`. Cette fonction retourne le chemin suivi ainsi que la distance parcourue entre `départ` et `arrivée` en utilisant l'algorithme décrit précédemment.

Écrire le programme principal permettant de déterminer un plus court chemin entre le sommet de départ 10 et le sommet d'arrivée 13. Le programme affichera le chemin suivi ainsi que la distance parcourue pour `w` = 0, `w` = 0.5 et `w` = 1.

5. Comment appelle-t-on les algorithmes lorsque `w` = 0, `w` = 0.5 et `w` = 1 ?

Analyse du problème

La fonction *h* est une fonction heuristique telle que $h(i)$ est le coût estimé du chemin qui reste à parcourir entre le sommet *i* et le sommet `arrivée`. On utilise la distance de Manhattan alors que dans l'exercice précédent « Algorithme A* » on utilise la distance euclidienne (ou distance à vol d'oiseau).

On considère différentes fonctions d'évaluation dans ce problème pour sélectionner un sommet :

- Algorithme glouton BFS (question 3) : la fonction d'estimation $f(i) = h(i)$ renvoie la distance de Manhattan entre le sommet *i* et le sommet `arrivée`.
- Algorithme de la question 4 : la fonction d'évaluation $f(i) = (1-w)\cdot g(i) + w \cdot h(i)$ renvoie une estimation de la distance entre le sommet de départ et le sommet d'arrivée en passant par un sommet *i*.

1.

```
dico={0:[1,5], 1 :[0,2,6], 2 :[1,3], 3 :[2,4], 4 :[3,9],\
  5 :[0,6,10], 6 :[1,5,11], 7 :[], 8 :[], 9 :[4,14],\
  10 :[5,11,15], 11 :[6,10], 12 :[], 13 :[14, 18],\
  14 :[9,13,19], 15 :[10,20], 16 :[], 17 :[],\
  18 :[13, 19, 23], 19 :[14,18, 24], 20 :[15,21],\
  21 :[20,22], 22 :[21,23], 23 :[18, 22,24], 24 :[19,23]}
```

Voir exercice 11.1 « Opérations de base sur les dictionnaires » dans le chapitre « Dictionnaire, pile, file, deque » pour l'utilisation des dictionnaires.

Le sommet 6 est relié aux sommets 1, 5 et 11. La valeur de la clé 6 est la liste des sommets adjacents [1, 5, 11].

Le nombre d'éléments du dictionnaire correspond au nombre *n* de sommets du graphe. On définit `nb_rues` le nombre de rues. Comme le nombre d'avenues est égal au nombre de rues, alors :

```
import math as m          # module math renommé m
nb_rues=int(m.sqrt(n))    # n = nombre de sommets
                          # = nombre de rues * nombre d'avenues
```

2. 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.

```
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
```

3. On définit une liste `SOMMETS` qui contient pour chaque sommet i :

- `SOMMETS[i][0]` : sommet précédent ;
- `SOMMETS[i][1]` : $g(i)$ = distance entre le sommet `départ` et le sommet i ;
- `SOMMETS[i][2]` : `True` si le sommet i est sélectionné, sinon `False` ;
- `SOMMETS[i][3]` : $f(i)$ = distance entre le sommet `départ` et le sommet `arrivée`.

La méthode gloutonne repose sur l'utilisation de sous-problèmes. Lorsqu'on est rendu au sommet `position`, on fait un choix local qui paraît être le meilleur en choisissant, dans la liste des sommets non sélectionnés, le sommet suivant `indice` dont la distance de Manhattan entre `indice` et `arrivée` est la plus petite. Ce choix n'est pas remis en cause ultérieurement. On privilégie les sommets « qui semblent » nous rapprocher de la destination.

a) Initialisation de l'algorithme :

On définit la liste `H` en utilisant la fonction `listeH` définie dans la question 2.

Tous les sommets sont non sélectionnés (`SOMMETS[i][2]=False`) sauf le sommet de départ. Pour ce sommet, la distance parcourue vaut 0.

Pour les sommets non sélectionnés, les distances parcourues depuis le sommet de départ sont initialisées à l'infini : `SOMMETS[i][1]=inf` et `SOMMETS[i][3]=inf`. La variable `position` prend initialement la valeur `départ`.

b) On définit une boucle `while(position!=arrivée)` tant que la variable `position` (sommet appartenant à la frontière entre les sommets sélectionnés et les sommets non sélectionnés) n'est pas égale à la variable `arrivée`.

- La boucle `for i in range(n)` avec le test `if SOMMETS[i][2]==False` permet de parcourir tous les sommets non sélectionnés. On calcule `g_i=SOMMETS[position][1]+1` (= distance entre `départ` et `position` + distance entre `position` et *i*) et `f_i=H[i]`.
- Si la nouvelle distance `f_i` entre `départ` et `arrivée` est inférieure à `SOMMETS[i][3]`, alors `SOMMETS[i][3] = f_i`. On met à jour également la distance entre le sommet `départ` et le sommet `i` : `SOMMETS[i][1] = g_i`. Le sommet `position` est le sommet précédent de `i` : `SOMMETS[i][0] = position`.
- On cherche le sommet `indice` (parmi les sommets non sélectionnés) pour lequel la distance *f* (`indice`) entre `départ` et `arrivée` est la plus petite.
- Ce sommet `indice` définit alors la nouvelle valeur de la variable `position` et ce sommet devient sélectionné.
- Si `indice` = `position`, on ne peut pas atteindre le sommet d'arrivée.

```
def init4(départ, n):  # n = nombre de sommets dans le graphe
    # la fonction initialise la liste SOMMETS à partir
    # de départ (int)
    SOMMETS=[]          # initialisation de la liste SOMMETS
    for i in range(n): # i varie entre 0 inclus et n exclu
        if i==départ:
            SOMMETS.append([-1, 0, True, 0])
           # la valeur -1 n'est pas utilisée
           # True : uniquement le sommet de départ est sélectionné
        else:
            SOMMETS.append([-1, inf, False, inf])
           # la valeur -1 n'est pas utilisée car distance infinie
           # False : sommet non sélectionné
    return SOMMETS
def BFS(dico, départ, arrivée):
    # la fonction permet d'avoir 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]
```

4. C'est le même algorithme que précédemment. On change la fonction d'évaluation : `f_i = (1-w)*g_i+w*h_i` au lieu de `f_i = h_i` pour sélectionner le sommet suivant lorsqu'on est rendu au sommet `position`. On retrouve l'algorithme de la question 2 avec `w = 1`.

```
def MANHATTAN (dico, départ, arrivée, w):
    # la fonction permet d'avoir un plus court chemin
    # de départ (int) à arrivée (int) dans la liste L
    # à partir du dictionnaire dico.
    # w est compris entre 0 et 1.
    # On récupère la liste SOMMETS
    n=len(dico)              # nombre de sommets
    H=listeH(arrivée, n)
```

```
    inf=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
                    h_i=H[i]  # coût estimé entre i et arrivée
                    # w=0:Dijkstra, w=0.5:algorithme A*, w=1:BFS
                    f_i=(1-w)*g_i+w*h_i
                    if f_i<SOMMETS[i][3]:
                        SOMMETS[i][1]=g_i
                            # distance départ->sommet i = g(i)
                        SOMMETS[i][3]=f_i
                            # distance départ->arrivée = f(i)
                        SOMMETS[i][0]=position
                            # sommet précédent sur le chemin
        # recherche du minimum des valeurs de f(i)
        # = distance départ->arrivée
        # pour les sommets non sélectionnés
        val_min=inf
        for i in range(n):
            if SOMMETS[i][2]==False and SOMMETS[i][3]<val_min:
                indice=i
                val_min=SOMMETS[i][3]  # f(i)
        if indice==position:
            return [],inf   # on n'atteint pas le sommet arrivée
        else:
            SOMMETS[indice][2]=True  # ce sommet est sélectionné
            position=indice          # nouvelle valeur de position

    # liste des sommets parcourus
    i=arrivée
    L=[arrivée]  # L = liste des sommets parcourus en sens inverse
    while (i!=départ):
        i=SOMMETS[i][0]
        L.append(i)
    L.reverse()  # il faut inverser la liste L pour obtenir la
                 # liste des sommets parcourus dans le sens direct
    return L, SOMMETS[arrivée][1]

# initialisation du programme
départ, arrivée=10, 13  # sommet de départ et sommet d'arrivée
L2, dist2=BFS(dico, départ, arrivée)
print('Algo BFS : chemin :', L2, '; distance', dist2)
w=0
for i in range(3):
    L2, dist2=MANHATTAN(dico, départ, arrivée, w)
    print('w =',w, '; chemin :', L2, '; distance', dist2)
    w=w+0.5
```

5. On retrouve plusieurs cas particuliers :

- `w` = 0 : algorithme de Dijkstra. L'algorithme devient une recherche en largeur sans stratégie d'exploration des sommets ;
- `w` = 0.5 : algorithme A* ;
- `w` = 1 : algorithme glouton BFS.

Si on souhaite aller de Bordeaux à Lyon, l'algorithme A* va plutôt explorer les sommets vers l'est qui ont une distance plus faible que les autres sommets alors que l'algorithme de Dijkstra fait une recherche en largeur sans stratégie d'exploration des sommets.

Le programme Python affiche les résultats suivants pour `départ` = 10 et `arrivée` = 13 :

- `w` = 0 ; chemin : [10, 15, 20, 21, 22, 23, 18, 13] ; distance = 7 ;
- `w` = 0.5 ; chemin : [10, 15, 20, 21, 22, 23, 18, 13] ; distance = 7 ;
- `w` = 1 ; chemin : [10, 11, 6, 1, 2, 3, 4, 9, 14, 13] ; distance = 9.

L'algorithme BFS (`w` = 1) ne donne pas la solution optimale : il privilégie le chemin passant par 11 qui semble plus près de 13 mais qui nécessite un long contournement pour atteindre effectivement 13.

Les algorithmes de Dijkstra et A* utilisent une heuristique dont le coût évalué (distance de Manhattan) est toujours inférieur ou égal au coût réel. On dit que ces heuristiques sont admissibles. On peut montrer que ces deux algorithmes retournent toujours une solution optimale si elle existe. Il peut y avoir plusieurs solutions optimales de même coût.

Dans la méthode gloutonne, on ne revient pas sur un choix local optimal alors que, avec la programmation dynamique, on peut revenir sur les choix précédents. L'algorithme est glouton pour toutes les valeurs de `w`.

L'algorithme de Floyd-Warshall (voir exercice 14.4 « Algorithme de Floyd-Warshall » dans le chapitre « Programmation dynamique ») permet de traiter des arcs dont le poids est négatif.

Partie 11

Programmation dynamique

Plan

14 Programmation dynamique (Spé) (sauf TSI et TPC)

Exercice 14.1 : Suite des nombres de Fibonacci, Top Down et Bottom Up

On note $(F_n)_{n\in\mathbb{N}}$ la suite des nombres de Fibonacci définie par : $F_0 = 0, F_1 = 1$, $\forall n \in \mathbb{N}, F_{n+2} = F_{n+1} + F_n$.

1. Écrire une fonction récursive `fibo1` qui permet de renvoyer le nombre de Fibonacci F_n. L'algorithme utilise-t-il la méthode « diviser pour régner » ?

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

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).

3. 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 ».

4. 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 ».

Analyse du problème

La méthode « diviser pour régner » permet de décomposer le problème initial en deux sous-problèmes.

Afin d'éviter de calculer plusieurs fois le même nombre de Fibonacci, on utilise la technique de mémoïsation.

Cours :

La méthode « diviser pour régner » peut se décomposer en trois étapes :

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

Cette méthode donne de très bons résultats dans de nombreux problèmes : dichotomie, tri par partition-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. On rencontre deux techniques :

- Technique récursive « Top Down » (de haut en bas) de mémoïsation. Lors d'un appel récursif, on regarde dans une liste intermédiaire si le sous-problème est déjà traité.
  Top Down : on résout dans le sens des données de grande taille vers les données de petite taille.
- Technique itérative « Bottom Up » (de bas en haut) : 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.

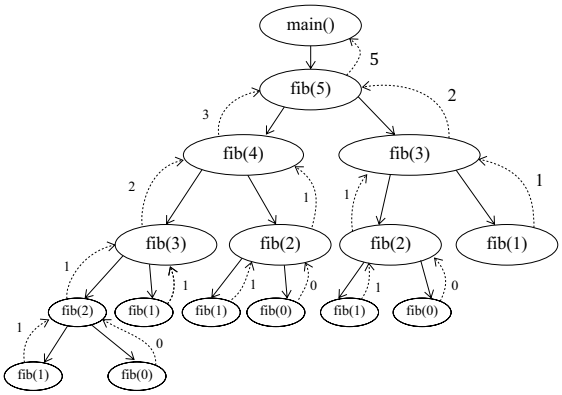
L'algorithme « Bottom Up » résout tous les sous-problèmes de taille inférieure alors que l'algorithme « Top Down » ne résout que les sous-problèmes de taille inférieure dont il a besoin.

1.

```
def fibo1(n):
    # la fonction renvoie le nombre de Fibonacci Fn
    # pour l'entier n
    if n==0:
        return 0 # condition d'arrêt
    elif n==1:
        return 1 # condition d'arrêt
    else:
        return (fibo1(n-1)+fibo1(n-2))
                  # appel récursif
```

L'algorithme est de type « diviser pour régner » puisqu'on décompose le problème (calcul de `fibo1(n)`) en deux sous-problèmes (calcul de `fibo1(n-1)` et `fibo1(n-2)`). On calcule récursivement chacun des deux sous-problèmes.

2. L'arbre ci-dessous représente les différents appels de la fonction `fibo1` que l'on note `fib`.



F_2 est recalculé 3 fois. F_3 est recalculé 2 fois.

L'inconvénient est que l'on augmente considérablement la complexité puisqu'on recalcule plusieurs la même valeur F_n.

3. 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. On dit que l'on a un chevauchement de sous-problèmes.

```
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))
```

4. La relation de récurrence s'écrit pour n ≥ 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]
```

Exercice 14.2 : Rendu de monnaie

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 cherche à rendre une certaine somme entière X en utilisant le moins de pièces, qui peuvent être identiques.

1. On utilise la méthode la plus intuitive qui consiste à commencer par rendre la plus grande pièce possible. Pour $X = 11$, on commence par rendre la pièce de 10.

On appelle $L[x]$ le nombre de pièces nécessaires pour rendre la somme x. La récurrence (1) peut s'écrire :

- $L[0] = 0$;
- si $x \geq 1$: $L[x] = 1 + L\left[x - S[i]\right]$ avec i le plus grand tel que $S[i] \leq x$.

On suppose que la liste S est triée par ordre croissant des valeurs. Écrire une fonction récursive `rendu1` qui admet comme arguments une liste `S` et un entier `X`. La fonction retourne le nombre de pièces nécessaires pour rendre la somme `X` en utilisant la récurrence (1).

L'algorithme utilise-t-il la méthode « diviser pour régner » ? Pourquoi cette méthode est-elle appelée gloutonne ? Est-ce que `rendu1([1, 4, 6], 8)` retourne la solution optimale ?

2. Pour trouver la solution optimale au rendu de monnaie, on utilise la récurrence (2) :

- $L[0] = 0$;
- si $x \geq 1$: $L[x] = 1 + \min\limits_{\substack{0 \leq i \leq n-1 \\ S_i \leq x}} L\left[x - S_i\right]$.

Écrire une fonction récursive `rendu2` qui admet comme arguments une liste `S` et un entier `X`. La fonction retourne le nombre minimal de pièces nécessaires pour rendre la somme `X`.

L'algorithme utilise-t-il la méthode « diviser pour régner » ?

Représenter l'arbre des appels de la fonction récursive `rendu2([1, 2, 5], 4)`. Quel est l'inconvénient ?

3. Écrire une fonction récursive `rendu3` qui admet comme arguments une liste `S`, un entier `X` et une liste `L` servant à stocker les résultats intermédiaires. La fonction retourne le nombre minimal de pièces nécessaires pour rendre la somme `X` en utilisant la programmation dynamique avec la récurrence (2).

Représenter l'arbre des appels de la fonction récursive `rendu3([1, 2, 5], 4, L)`.

Utilise-t-on la technique « Top Down » (de haut en bas) ou « Bottom Up » (de bas en haut) dans la fonction `rendu3` ?

4. Écrire une fonction itérative `rendu4` qui admet comme arguments une liste `S`, un entier `X` et une liste `L`. La fonction retourne le nombre minimal de pièces nécessaires pour rendre la somme `X`. On part de la plus petite somme possible à rendre et on calcule les éléments suivants de `L` en utilisant la récurrence (2).

Utilise-t-on la technique « Top Down » ou « Bottom Up » ?

5. On souhaite reconstruire la solution optimale à partir de l'information calculée, c'est-à-dire obtenir la liste des pièces utilisées. On définit la liste T :

- $T[x] > 0$ si on a utilisé la pièce d'indice $T[x]$ pour rendre la somme x dans la récurrence (2) ;
- $T[x] = 0$ sinon.

On définit une liste `PIECES` initialement vide. On considère une boucle `while` en partant de $x = X$:

- $T[x]$ désigne l'indice de la pièce utilisée pour rendre la somme x dans la récurrence (2) ;
- ajouter dans la liste `PIECES` la valeur de la pièce utilisée ;
- retrancher cette valeur à x.

Modifier la fonction `rendu3` qui utilise la technique « Top Down » pour obtenir la liste des pièces utilisées.

6. Modifier la fonction `rendu4` qui utilise la technique « Bottom Up » pour obtenir la liste des pièces utilisées.

Analyse du problème

On étudie plusieurs algorithmes permettant d'optimiser le rendu de monnaie. La méthode « diviser pour régner » permet de décomposer le problème initial en deux sous-problèmes. La programmation dynamique permet d'obtenir une solution optimale en utilisant deux techniques : « Top Down » et « Bottom Up ».

On verra la différence entre la méthode gloutonne et la programmation dynamique.

1. On suppose que la liste S est triée par ordre croissant des valeurs.

```
def rendu1(S, X):
    # la fonction renvoie le nombre de pièces nécessaires
    # pour rendre la somme X en utilisant la liste S :
    # S[i] = valeur de la pièce d'indice i
    if X==0:  # condition d'arrêt
        return 0
    else:
            # recherche de i le plus grand tel que S[i] <= X
        i=len(S)-1
        while S[i]>X:
            i=i-1
            # ajoute 1 au nombre de pièces ;
```

```
            # puisqu'on utilise la pièce S[i]
            # il reste donc à rendre la monnaie à X - S[i]
        return 1+rendu1(S, X-S[i]) # appel récursif

S=[1, 4, 6]
X=8
print(rendu1(S, X))                # on obtient : 3
```

Cours :
La méthode « diviser pour régner » peut se décomposer en trois étapes :

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

Dans la méthode gloutonne (greedy en anglais), on effectue une succession de choix, chacun d'eux semble être le meilleur sur le moment. On résout alors le sous-problème mais on ne revient jamais sur le choix déjà effectué.

L'algorithme est de type « diviser pour régner » puisqu'on décompose le problème (calcul de $L[X]$) en un sous-problème (calcul de $L[X - S[i]]$ avec i le plus grand tel que $S[i] \leq X$). On calcule récursivement le sous-problème.

À chaque étape de l'algorithme, on commence par rendre la plus grande pièce possible, c'est-à-dire la plus grande pièce dont la valeur est inférieure à la somme à rendre. C'est la solution qui semble être la meilleure et la plus intuitive. On déduit alors de cette pièce la somme à rendre et on est ramené à un sous-problème avec une somme à rendre plus petite. On recommence jusqu'à obtenir une somme nulle.

Cet algorithme est très simple mais à chaque étape on n'étudie pas tous les cas possibles puisqu'on se contente de choisir la pièce la plus grande que l'on peut rendre.

Dans le cas où $S = [1, 4, 6]$ et $X = 8$, on n'obtient pas la solution optimale. L'algorithme glouton (greedy algorithm, en anglais) renvoie 3 (1 pièce de 6 et 2 pièces de 1) alors que la solution optimale est 2 (2 pièces de 4).

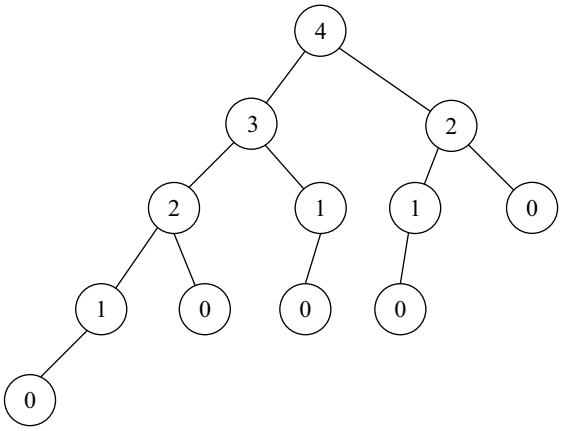
2.

```
def rendu2(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:
        mini=X        # recherche du minimum de X-S[i]
        for i in range(len(S)):  # i varie entre 0 inclus
                                 # et len(S) exclu
            if S[i]<=X:          # il faut que Si <= X
                res=rendu2(S, X-S[i])
                                 # appel récursif pour
                                 # rendre la monnaie à X-S[i]
```

```
                if res<mini:
                    mini=res
        return 1+mini            # ajoute 1 au nombre de pièces

S=[1, 4, 6]
X=8
print(rendu2(S, X))  # on obtient 2, c'est-à-dire la solution
                     # optimale alors que rendu1(S, X)
                     # retourne 3
```

L'algorithme est de type « diviser pour régner » puisqu'on décompose le problème (calcul de $L[X]$) en plusieurs sous-problèmes (calcul de $L[X–S[0]]$, $L[X–S[1]]$,..., $L[X–S[n\text{-}1]]$). On combine les différents sous-problèmes pour résoudre le problème de départ.

L'arbre ci-dessous représente les différents appels de la fonction `rendu2([1, 2, 5], 4)` qui retourne 2.



On appelle `rendu2` pour $X = 4$.

- On appelle `rendu2` pour $X–S[i] = 4 – 1 = 3$ avec $i = 0$ (de la boucle `for`).
- On appelle `rendu2` pour $3 – 1 = 2$ avec $i = 0$ (de la boucle `for`).
- On appelle `rendu2` pour $2 – 1 = 1$ avec $i = 0$ (de la boucle `for`). On appelle `rendu2` pour $1 – 1 = 0$ avec $i = 0$. On arrive à la condition d'arrêt.
- On dépile et on revient à l'appel pour $X = 2$ avec $i = 1$ (de la boucle `for`). On appelle `rendu2` pour $2 – 2 = 0$. On arrive à la condition d'arrêt.
- On dépile et on revient à l'appel pour $X = 3$ avec $i = 1$ (de la boucle `for`). On appelle `rendu2` pour $3 – 2 = 1$. On arrive à la condition d'arrêt.
- On dépile et on revient à l'appel pour $X = 4$ avec $i = 2$ (de la boucle `for`). On appelle `rendu2` pour $4 – 2 = 2$.
- ...

On constate que l'on calcule plusieurs fois le nombre de pièces à rendre pour $X = 2$. Les sous-problèmes ne sont pas indépendants. On est amené à résoudre plusieurs fois le même sous-problème. On dit que l'on a un chevauchement de sous-problèmes.

3. Pour éviter de calculer plusieurs fois le nombre de pièces à rendre pour une valeur de *X*, on garde en mémoire le résultat dans la liste `L`. La liste `L` doit contenir les valeurs suivantes : 0, 1, 2, ..., *X*.

```
def rendu3(S, X, L): # programmation dynamique -
                     # technique Top Down
    # 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
    # stockage des résultats intermédaires dans L :
    # L[i] = nombre de pièces nécessaires pour
    # rendre la somme i
    if X==0:          # condition d'arrêt
        return 0
    elif L[X]>0:      # valeur déjà calculée
        return L[X]   # retourne le nombre de pièces nécessaires
                      # pour rendre la somme X
    else:
        mini=X        # recherche du minimum de X-S[i]
        for i in range(len(S)):  # i varie entre 0 inclus
                                 # et len(S) exclu
            if S[i]<=X:          # il faut que Si <= X
                res=rendu3(S, X-S[i], L)
                                 # appel récursif pour
                                 # rendre la monnaie à X-S[i]
                if res<mini:
                    mini=res
        L[X]=1+mini              # technique de mémoïsation
        return 1+mini            # ajoute 1 au nombre de pièces

L=[0 for i in range(X+1)]        # initialisation - liste L
                                 # avec des valeurs nulles
print(rendu3(S, X, L))           # on obtient 2
```

Remarque : On peut utiliser un dictionnaire au lieu de la liste `L`. Pour chaque élément du dictionnaire `dico`, on précise la clé et la valeur associée. La clé est l'entier `i` et la valeur est `dico[i]` = nombre de pièces nécessaires pour rendre la somme `i`.

```
def rendu3_dico(S, X, dico): # programmation dynamique -
                             # technique Top Down
    # 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
    # stockage des résultats intermédaires dans dico :
    # dico[i] = nombre de pièces nécessaires pour
    # rendre la somme i
    if X==0:             # condition d'arrêt
        return 0
```

```
    elif X in dico:      # valeur déjà calculée
        return dico[X]   # retourne le nombre de pièces
                         # nécessaires pour rendre la somme X
    else:
        mini=X           # recherche du minimum de X-S[i]
        for i in range(len(S)):  # i varie entre 0 inclus
                                 # et len(S) exclu
            if S[i]<=X:          # il faut que Si <= X
                res=rendu3_dico(S, X-S[i], dico)
                                 # appel récursif pour
                                 # rendre la monnaie à X-S[i]
                if res<mini:
                    mini=res
        dico[X]=1+mini           # technique de mémoïsation
        return 1+mini            # ajoute 1 au nombre de pièces

dico={}
S=[1, 4, 6]
X=8
print(rendu3_dico(S, X, dico))
```

Cours :

La programmation dynamique est souvent utilisée pour résoudre des problèmes d'optimisation. Elle comprend différentes étapes :

- Recherche d'une récurrence pour déterminer la valeur d'une solution optimale.
- Utilisation de la technique Top Down ou Bottom Up.

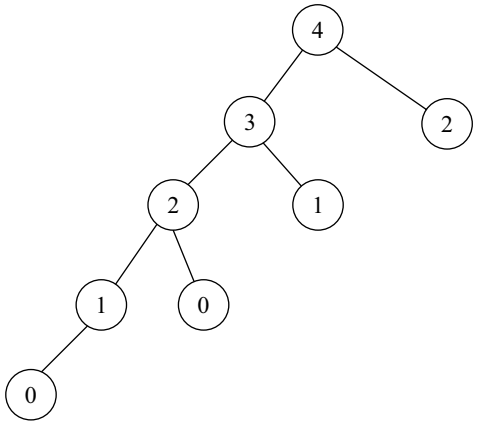
La programmation dynamique et la méthode gloutonne reposent sur l'utilisation de sous-problèmes. Il y a une différence importante :

- 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.

L'arbre ci-dessous représente les différents appels de la fonction `rendu3([1, 2, 5], 4)`.



On appelle `rendu3` pour $X = 4$.

- On appelle `rendu3` pour $X–S[i] = 4 – 1 = 3$ avec $i = 0$ (de la boucle `for`).
- On appelle `rendu3` pour $3 – 1 = 2$ avec $i = 0$ (de la boucle `for`).
- On appelle `rendu3` pour $2 – 1 = 1$ avec $i = 0$ (de la boucle `for`). On appelle `rendu3` pour $1 – 1 = 0$ avec $i = 0$. On arrive à la condition d'arrêt.
- On dépile et on revient à l'appel pour $X = 2$ avec $i = 1$ (de la boucle `for`). On appelle `rendu3` pour $2 – 2 = 0$. On arrive à la condition d'arrêt.
- On dépile et on revient à l'appel pour $X = 3$ avec $i = 1$ (de la boucle `for`). On appelle `rendu3` pour $3 – 2 = 1$. On arrive à la condition d'arrêt.
- On dépile et on revient à l'appel pour $X = 4$ avec $i = 2$ (de la boucle `for`). On appelle `rendu3` pour $4 – 2 = 2$. On arrive à la condition d'arrêt puisque $L[2]$ a déjà été calculé.

On ne calcule pas plusieurs fois le nombre de pièces à rendre pour $X = 2$.

La fonction `rendu3` 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.

4. On a deux boucles `for` imbriquées. Il faut remplir toute la liste L avant d'obtenir la valeur optimale ($x = X$). On remarque que la récurrence (2) s'écrit avec x (pour la fonction `rendu4`) et non X (pour la fonction `rendu3`).

```
def rendu4(S, X, L):   # programmation dynamique –
                       # technique Bottom Up
    # 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
    # stockage des résultats intermédaires dans L :
    # L[i] = nombre de pièces nécessaires pour
    # rendre la somme i
    for x in range(1, X+1):      # x varie entre 1 inclus
                                 # et X+1 exclu
        mini=X
```

```
        for i in range(len(S)):  # i varie entre 0 inclus
                                 # et len(S) exclu
            if S[i]<=x and L[x-S[i]]<mini:
                mini=L[x-S[i]]
        L[x]=1+mini
    return L[X]

L=[0 for i in range(X+1)]        # initialisation - liste L
                                 # avec des valeurs nulles
print(rendu4(S, X, L))           # on obtient 2
```

L'algorithme `rendu4` utilise la programmation dynamique avec la technique « Bottom Up » : on utilise la même formule de récurrence mais on part de la plus petite valeur à rendre au lieu de partir de la plus grande valeur à rendre (technique « Top Down »).

Dans la fonction `rendu4`, on incrémente *X* de 1 inclus à *X* inclus (ou *X*+1 exclu).

- 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 (deux boucles `for` imbriquées). Il faut d'abord remplir entièrement la liste `L` avant de retourner la solution optimale au problème.

5. En dessous de la ligne `L[X]=1+mini`, on ajoute la ligne `T[X]=indice` afin de préciser l'indice de la pièce utilisée pour rendre la somme *X*. La variable `indice` est définie lors du test `if res>mini:`.

```
def rendu3_piece(S, X, L, T):   # programmation dynamique -
                                # technique Top Down
    # 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
    # stockage des résultats intermédaires dans L :
    # L[i]=nombre de pièces nécessaires pour
    # rendre la somme i
    # T[x] > 0 si on a utilisé la pièce d'indice T[x]
    # pour rendre la somme x
    if X==0:              # condition d'arrêt
        return 0
    elif L[X]>0:          # valeur déjà calculée
        return L[X]       # retourne le nombre de pièces
                          # nécessaires pour rendre
                          # la somme X
    else:
        mini=X            # recherche du minimum de X-S[i]
        for i in range(len(S)):  # i varie entre 0 inclus
                                 # et len(S) exclu
            if S[i]<=X:          # il faut que Si <= X
                res=rendu3_piece(S, X-S[i], L, T)
                                 # appel récursif pour
                                 # rendre la monnaie à X-S[i]
```

```
                if res<mini:
                    mini=res
                    indice=i
        L[X]=1+mini              # technique de mémoïsation
        T[X]=indice              # indice de la pièce utilisée
                                 # pour rendre la somme X
        return 1+mini            # ajoute 1 au nombre de pièces

S=[1, 2, 5]
X=4
L=[0 for i in range(X+1)]        # initialisation - liste L
                                 # avec des valeurs nulles
T=[0 for i in range(X+1)]        # initialisation - liste T
                                 # avec des valeurs nulles
print(rendu3_piece(S, X, L, T))

PIECES=[]
x=X
while x>0:
    PIECES.append(S[T[x]])
       # T[x] = indice de la pièce utilisée pour
       # rendre la somme x
       # S[T[x]] = valeur de la pièce d'indice T[x]
    x=x-S[T[x]]
       # on retranche la valeur S[T[x]] à x pour
       # chercher les autres pièces
print(PIECES)
```

Le programme Python affiche :

```
2
[4, 4]
```

6. En dessous de la ligne `L[x]=1+mini`, on ajoute la ligne `T[x]= indice` afin de préciser l'indice de la pièce utilisée pour rendre la somme *x*. La variable `indice` est définie lors du test `if S[i]<=x and L[x-S[i]]<mini:`.

On remarque que l'on utilise *x* (`rendu4_piece`) et non *X* (`rendu3_piece`).

```
def rendu4_piece(S, X, L, T):   # programmation dynamique -
                                # technique Bottom Up
    # la fonction renvoie le nombre de pièces nécessaires
    # pour rendre la somme X ainsi que la liste des pièces
    # en utilisant la liste S
    # stockage des résultats intermédaires dans L :
    # L[i] = nombre de pièces nécessaires pour
    # rendre la somme i
    # T[x] > 0 si on a utilisé la pièce d'indice T[x]
    # pour rendre la somme x
    for x in range(1, X+1):      # x varie entre 1 inclus
                                 # et X+1 exclu
        mini=X
        indice=-1
        for i in range(len(S)):  # i varie entre 0 inclus
```

```
                                # et len(S) exclu
            if S[i]<=x and L[x-S[i]]<mini:
                mini=L[x-S[i]]
                indice=i        # indice de la pièce
        L[x]=1+mini
        T[x]=indice             # indice de la pièce utilisée
                                # pour rendre la somme x
    # création de la liste des pièces utilisées
    # pour rendre la monnaie
    PIECES=[]
    x=X
    while x>0:
        PIECES.append(S[T[x]])
           # T[x] = indice de la pièce utilisée pour
           # rendre la somme x
           # S[T[x]] = valeur de la pièce d'indice T[x]
        x=x-S[T[x]]
           # on retranche la valeur S[T[x]] à x pour
           # chercher les autres pièces
    return L[X], PIECES

S=[1, 2, 5]
X=4
L=[0 for i in range(X+1)]       # initialisation - liste L
                                # avec des valeurs nulles
T=[0 for i in range(X+1)]       # initialisation - liste T
                                # avec des valeurs nulles
print(rendu4_piece(S, X, L, T))
```

Le programme Python affiche :

```
2
[4, 4]
```

Exercice 14.3 : Problème du sac à dos

On considère un sac à dos dont la masse maximale est notée M. On cherche à maximiser la valeur totale des objets insérés dans le sac à dos. On dispose de n objets modélisés par la liste de listes `S` :

- `S[i][0]` désigne la valeur de l'objet d'indice i notée v_i (i varie de 0 à n–1).
- `S[i][1]` désigne la masse de l'objet d'indice i notée m_i (i varie de 0 à n–1).

On suppose dans tout le problème que $\sum_{i=0}^{n-1} m_i > M$ et que les masses sont des entiers. Le premier objet de la liste `S` a pour indice 0.

1. On utilise la méthode intuitive consistant à insérer au fur et à mesure les objets qui ont le plus grand rapport valeur/masse. On suppose que la liste `S` est triée par ordre décroissant du 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.

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 ?

2. On considère `L` la liste telle que `L[m][i]` désigne la valeur maximale des objets que l'on peut insérer dans le sac à dos de masse maximale *m* en ne considérant que les *i* premiers objets de la liste *S* (indices des objets *S* compris entre 0 et *i*–1).

Pour trouver la solution optimale au problème du sac à dos, on utilise la récurrence suivante (1) :

- `L[m][0]` = 0.
- `L[m][i]` = `L[m][i-1]` si `S[i-1][1]` > *m* et *i* > 0.
- `L[m][i]` = `max(L[m][i-1], S[i-1][0]+L[m-S[i-1][1],i-1])` si `S[i-1][1]` ≤ *m* et *i* > 0.

Écrire une fonction récursive `algo2` qui admet comme arguments une liste `S`, un entier `M` et un entier `i` (on ne considère que les `i` premiers objets de la liste `S`). On n'utilisera pas dans cette question la liste `L` pour stocker les résultats intermédiaires. La fonction retourne la valeur maximale des objets que l'on peut insérer dans le sac à dos.

Démontrer la terminaison de la fonction `algo2`. L'algorithme utilise-t-il la méthode « diviser pour régner » ? Quel est le principal inconvénient de cet algorithme ?

3. Écrire une fonction récursive `algo3` qui admet comme arguments une liste `S`, un entier `M`, un entier `i` (on ne considère que les `i` premiers objets de la liste `S`) et une liste `L` (stockage des résultats intermédiaires). La fonction retourne la valeur maximale des objets que l'on peut insérer dans le sac à dos en utilisant la programmation dynamique. Utilise-t-on la technique « Top Down » (de haut en bas) ou « Bottom Up » (de bas en haut) ?

4. Écrire une fonction itérative `algo4` qui admet comme arguments une liste `S`, un entier `M` et une liste `L` (stockage des résultats intermédiaires). La fonction retourne la valeur maximale des objets que l'on peut insérer dans le sac à dos en utilisant la programmation dynamique avec la récurrence (1). Les résultats intermédiaires sont stockés dans la liste `L`.

On calcule les éléments de `L` en partant de la plus petite valeur de `i` et de la plus petite valeur de `m`.

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 ?

5. On souhaite reconstruire la solution optimale à partir de l'information calculée, c'est-à-dire connaître la liste des objets insérés dans le sac. On définit la liste `T` :

- `T[m][i]` = 1 si on a inséré le $i^{\text{ème}}$ objet (indice i–1 dans S) dans le sac à dos de masse maximale m ;
- `T[m][i]` = 0 sinon.

On définit une liste `OBJETS` initialement vide. On pose $m = M$. On considère une boucle `for` en partant de $i = n$:

- Si `T[m][i]` = 1, alors on a inséré le $i^{\text{ème}}$ objet (indice i–1 dans S) dans le sac à dos. Ajouter cet objet dans la liste `OBJETS` et retrancher la masse de cet objet à la masse m.

Modifier la fonction `algo3` qui utilise la technique « Top Down » pour qu'elle retourne la valeur maximale ainsi que la liste des objets insérés.

6. Modifier la fonction `algo4` qui utilise la technique « Bottom Up » pour qu'elle retourne la valeur maximale ainsi que la liste des objets insérés.

Analyse du problème

On étudie plusieurs méthodes permettant de maximiser la valeur des objets insérés dans un sac à dos. La programmation dynamique permet d'obtenir une solution optimale en utilisant deux techniques : « Top Down » et « Bottom Up ». Une liste temporaire permet de stocker les résultats des sous-problèmes. On verra la différence entre les deux techniques concernant le nombre de sous-problèmes à traiter.

1.

```
def algo1(S, M):   # S 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.

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.

2. On cherche à insérer le $i^{\text{ème}}$ objet. On ne fait plus le même test que dans la question précédente (nouvelle masse totale $\leq M$) mais on teste si la masse `S[i-1][1]` du $i^{\text{ème}}$ objet (indice i–1) est inférieure à M puisqu'on traite le cas d'un sac à dos de masse maximale `M-S[i-1][1]`.

On définit la fonction récursive `algo2` :

- `algo2(S, M, i)` retourne la valeur maximale des objets dans le sac à dos de masse maximale `M` en ne considérant que les i premiers objets de la liste S (indices compris entre 0 et i–1).
- `algo2(S, M, i-1)` : valeur maximale dans le sac à dos de masse maximale `M` en ne considérant que les i–1 premiers objets.
- `algo2(S, M-S[i-1][1], i-1)`: valeur maximale dans le sac à dos de masse maximale `M-S[i-1][1]` en ne considérant que les i–1 premiers objets.

On considère deux cas :

- Si l'objet d'indice i–1 a une masse supérieure à M, alors on ne peut pas l'insérer et le programme retourne `algo2(S, M, i-1)` puisqu'il suffit de considérer les i–1 premiers objets.
- Sinon, on cherche le maximum de deux nombres :
  - `algo2(S, M, i-1)` en n'insérant pas l'objet d'indice i–1 ;
  - valeur de l'objet d'indice `(i-1) + algo2(S, M-S[i-1][1], i-1)` en insérant l'objet d'indice i–1. Il faut en effet considérer la valeur maximale dans le sac à dos de masse `M-S[i-1][1]` avec les i–1 premiers objets. En faisant la somme des deux valeurs, la masse

maximale du sac à dos vaut toujours `S[i-1][1]+(M-S[i-1][1])=M`.

```
def algo2(S, M, i):
    # la fonction retourne la valeur des objets que l'on peut
    # insérer avec une masse maximale M
    # et S liste de listes avec [valeur, masse]
    # on ne considère que les i premiers objets de la liste S
    if i==0:          # condition d'arrêt
        return 0      # pas d'objet à insérer dans le sac à dos
    elif S[i-1][1]>M: # l'objet d'indice i-1 est de masse > M
                      # on ne peut pas l'insérer
        return algo2(S, M, i-1)  # appel récursif
    else:
        return max(algo2(S, M, i-1),\
                   S[i-1][0]+algo2(S, M-S[i-1][1], i-1))
                      # appel récursif

M=30
S=[[15, 6], [60, 25], [10, 5], [7, 8], [10, 20]]
    # [valeur, masse]
print('ALGO2 :', algo2(S, M, len(S)))
```

Le programme Python retourne `70`, qui est la solution optimale.

On considère le variant de boucle *i*. 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 $i \geq 0$.

L'algorithme est de type « diviser pour régner » puisqu'on décompose le problème (`algo2(S, M, i)`) en plusieurs sous-problèmes (`algo2(S, M, i-1)` et `algo2(S, M-mi, i-1)`). On combine les différents sous-problèmes pour résoudre le problème de départ.

Le principal inconvénient est que l'on calcule plusieurs la même valeur totale. 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.

3. Pour éviter de calculer plusieurs fois la même valeur totale, on garde en mémoire le résultat dans la liste `L` : `L[m ][i]` contient la valeur maximale des objets dans le sac à dos de masse maximale *m* en ne considérant que les *i* premiers objets.

La liste `L` doit contenir `M+1` lignes et `len(S)+1` colonnes.

```
def algo3(S, M, i, L):
    # la fonction retourne la valeur des objets que l'on peut
    # insérer avec une masse maximale M
    # et S liste de listes avec [valeur, masse]
    # on ne considère que les i premiers objets de la liste S
    # stockage des résultats intermédaires dans la liste L
    if L[M][i]>0:      # le sous-problème a déjà été traité
        return L[M][i] # retourne la valeur déjà calculée
    elif i==0:         # condition d'arrêt
```

```
        return 0       # pas d'objet à insérer dans
                       # le sac à dos
    elif S[i-1][1]>M:
        total=algo3(S, M, i-1, L)  # appel récursif
        L[M][i]=total  # l'objet d'indice i-1 est de masse > M
                       # on ne peut pas l'insérer
        return total
    else:
        total=max(algo3(S, M, i-1, L),\
                  S[i-1][0]+algo3(S,M-S[i-1][1], i-1, L))
                       # appel récursif
        L[M][i]=total
        return total

M=30
S=[[15, 6], [60, 25], [10, 5], [7, 8], [10, 20]]
    # [valeur, masse]
L=[[0 for j in range(len(S)+1)] for i in range(M+1)]
print('ALGO3 :', algo3(S, M, len(S), L))
```

La fonction `algo3` 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.

4. On a deux boucles `for` imbriquées.

La boucle `for i` commence à $i = 1$ puisqu'on considère $i-1$ dans la récurrence (1).

La boucle `for m` commence à $m = 0$ et se termine à M inclus. Il faut remplir toute la liste `L` avant d'obtenir la valeur optimale (i = `len(S)` et $m = M$).

On remarque que la récurrence (1) s'écrit avec m (`algo4`) et non M (`algo3`).

```
def algo4(S, M, L):
    # la fonction retourne la valeur des objets que l'on peut
    # insérer avec une masse maximale M
    # et S liste de listes avec [valeur, masse]
    # stockage des résultats intermédaires dans la liste L
    for i in range(1, len(S)+1):  # i varie entre 1 inclus
                                  # et len(S)+1 exclu
        for m in range(M+1):      # m varie entre 0 inclus
                                  # et M+1 exclu
            if S[i-1][1]>m:       # on considère l'objet
                                  # d'indice i-1
                L[m][i]=L[m][i-1] # l'objet d'indice i-1 est
                                  # de masse > m
                                  # on ne peut pas l'insérer
            else:
                L[m][i]=max(L[m][i-1],\
                            S[i-1][0]+L[m-S[i-1][1]][i-1])
    return L[M][len(S)]
```

La fonction `algo4` utilise la programmation dynamique avec la technique « Bottom Up » : on utilise la même formule de récurrence que dans `algo2`

et `algo3` mais on part du plus petit nombre d'objets à insérer au lieu de partir du plus grand nombre d'objets à insérer (technique « Top Down »).

Il faut deux boucles `for` imbriquées pour faire varier le nombre d'objets à insérer et la masse maximale du sac à dos.

- 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 (deux boucles `for` imbriquées). Il faut d'abord remplir entièrement la liste `L` avant de retourner la solution optimale au problème.

```
L=[[0 for j in range(len(S)+1)] for i in range(M+1)]
print('ALGO4 :', algo4(S, M, L))
```

5. Lors de la recherche du maximum dans `algo3`, il faut savoir si l'objet d'indice *i*–1 a été inséré ou non. Si on insère l'objet d'indice *i*–1, alors `T[M][i]` = 1.

```
def algo3_objets(S, M, i, L, T):
    # la fonction retourne la valeur des objets que l'on peut
    # insérer avec une masse maximale M
    # et S liste de listes avec [valeur, masse]
    # on ne considère que les i premiers objets de la liste S
    # stockage des résultats intermédaires dans la liste L
    # T[m][i] = 1 si on a inséré le ième objet
    # (indice i-1 dans S) dans le sac à dos
    # de masse maximale m
    if L[M][i]>0: # le sous-problème a déjà été traité
        return L[M][i] # retourne la valeur déjà calculée
    elif i==0:         # condition d'arrêt
        return 0       # pas d'objet à insérer dans
                       # le sac à dos
    elif S[i-1][1]>M:
        total=algo3_objets(S, M, i-1, L, T)  # appel récursif
        L[M][i]=total  # l'objet d'indice i-1 est de masse > M
                       # on ne peut pas l'insérer
        return total
    else:
        a=algo3_objets(S, M, i-1, L, T)
        b=S[i-1][0]+algo3_objets(S, M-S[i-1][1], i-1, L, T)
        if a > b:
            L[M][i]=a  # on n'a pas inséré l'objet d'indice i-1
            total=a
        else:
            L[M][i]=b
            T[M][i]=1  # on a inséré le ième objet dans le
                       # sac à dos de masse M
            total=b
        return total

M=30                   # entier
S=[[15, 6], [60, 25], [10, 5], [7, 8], [10, 20]]
    # [valeur, masse]
```

```
L=[[0 for j in range(len(S)+1)] for i in range(M+1)]
T=[[0 for j in range(len(S)+1)] for i in range(M+1)]
print('algo3_objets :', algo3_objets(S, M, len(S), L, T))
OBJETS=[]
m=M
for i in range(len(S), 0, -1):  # i varie entre len(S) inclus
                                # et 0 exclu avec pas = -1
    if T[m][i]==1:    # i désigne le ième objet
        OBJETS.append(S[i-1])   # on a inséré le ième objet
                                # d'indice i-1
        m=m-S[i-1][1] # retranche à m la masse de l'objet inséré
print(OBJETS)
```

Pour obtenir la liste des objets insérés, il faut créer une liste `OBJETS` vide et considérer une boucle `for` en partant de `len(S)`. Si le $i^{ème}$ objet d'indice i–1 a été inséré, on l'ajoute dans la liste `OBJETS`. Il faut ensuite retrancher à m la masse de cet objet inséré et tester `T[m][i]` avec la nouvelle masse et le nouvel indice.

Le programme principal affiche :

```
70                  # valeur totale du sac à dos
[[10, 5], [60, 25]] # on a ajouté 1 objet de valeur 10
                    # et une autre de valeur 60
```

6. Lors de la recherche du maximum dans `algo4`, il faut savoir si l'objet d'indice i–1 a été inséré ou non. Si on insère l'objet d'indice i–1, alors `T[m][i]` =1.

Il s'agit bien de `T[m][i]` et non de `T[M ][i]` puisqu'on remplit la liste pour toutes les valeurs de m et de i.

```
def algo4_objets(S, M, L, T):
    # la fonction retourne la valeur des objets que l'on peut
    # insérer avec une masse maximale M
    # et S liste de listes avec [valeur, masse]
    # stockage des résultats intermédaires dans la liste L
    # T[m][i] = 1 si on a inséré le ième objet
    # (indice i-1 dans S) dans le sac à dos
    # de masse maximale m
    for i in range(1, len(S)+1):   # i varie entre 1 inclus
                                   # et len(S)+1 exclu
        for m in range(M+1):       # m varie entre 0 inclus
                                   # et M+1 exclu
            if S[i-1][1]>m:        # on considère l'objet
                                   # d'indice i-1
                L[m][i]=L[m][i-1]  # l'objet d'indice i-1 est
                                   # de masse > m
                                   # on ne peut pas l'insérer
            else:
                a=L[m][i-1]
                b=S[i-1][0]+L[m-S[i-1][1]][i-1]
                if a > b:
```

```
                    L[m][i]=a       # on n'a pas inséré l'objet
                                    # d'indice i-1
                else:
                    L[m][i]=b
                    T[m][i]=1       # on a inséré le ième objet
                                    # dans le sac à dos
                                    # de masse m
    # création de la liste des objets insérés dans le sac à dos
    OBJETS=[]
    m=M
    for i in range(len(S), 0, -1): #i désigne le ième objet
        if T[m][i]==1:
            OBJETS.append(S[i-1])  # on a inséré le ième objet
                                    # d'indice i-1
            m=m-S[i-1][1]          # on retranche à m la masse
                                    # de l'objet inséré
    return L[M][len(S)], OBJETS

M=30                                # entier
S=[[15, 6], [60, 25], [10, 5], [7, 8], [10, 20]]
    # [valeur, masse]
L=[[0 for j in range(len(S)+1)] for i in range(M+1)]
T=[[0 for j in range(len(S)+1)] for i in range(M+1)]
print('algo4_objets :', algo4_objets(S, M, L, T))
```

Le programme Python affiche :

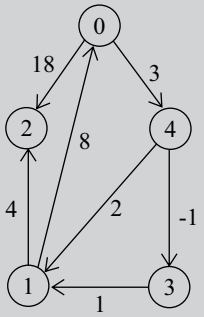
```
70
[[10, 5], [60, 25]]
```

Exercice 14.4 : Algorithme de Floyd-Warshall

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 $\left(\mathbf{M}_{i,j}\right)_{0\leq i,j\leq n-1}$ 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 $d_0(i,j)=\mathbf{M}[i][j]$.
- 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`.

1. Construire la matrice d'adjacence $\left(\mathbf{M}_{i,j}\right)_{0\leq i,j\leq n-1}$ du graphe G, définie par :
Pour tous les indices i, j, $\mathbf{M}_{i,j}$ 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.

2. 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 » ?

3. 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 » ?

4. 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 » ?

5. 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 $-\infty <$ DIST[i][j][k] $< \infty$, alors 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.
- PRECEDENT$[i][j][0] = i$ si $i \neq j$ et $\mathbf{M}[i][j] < \infty$.
- La valeur de PRECEDENT$[i][j][k]$ lorsque $i = j$ ou DIST$[i][j][k] = \infty$ ou DIST$[i][j][k] = -\infty$ n'a aucune importance. On peut la prendre égale à –1.
- PRECEDENT$[i][j][k]$ = PRECEDENT$[i][j][k-1]$ si $d_k(i,j) = d_{k-1}(i,j)$.
- PRECEDENT$[i][j][k]$ = PRECEDENT$[k-1][j][k-1]$ si

$$d_k(i,j) = d_{k-1}(i,k-1) + d_{k-1}(k-1,j) \text{ et } d_k(i,j) \neq \infty$$
$$\text{et PRECEDENT}[k-1][j][k-1] \neq -1$$

Écrire le programme principal permettant d'afficher le chemin suivi entre les sommets `départ` et `arrivée`.

Analyse du problème

On considère des graphes orientés contenant des arêtes de poids négatif et n'ayant pas de cycle de poids négatif. Le graphe n'est pas nécessairement fortement connexe.

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.

1. La matrice d'adjacence est : $\mathbf{M} = \begin{pmatrix} 0 & \infty & 18 & \infty & 3 \\ 8 & 0 & 4 & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \\ \infty & 1 & \infty & 0 & \infty \\ \infty & 2 & \infty & -1 & 0 \end{pmatrix}$.

Comme le graphe est orienté, la matrice n'est pas nécessairement symétrique.

Cours :

La méthode « diviser pour régner » peut se décomposer en trois étapes :

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

2. L'algorithme est de type « diviser pour régner » puisqu'on décompose le problème (calcul de $d_k(i,j)$) 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)
```

Cours :

On rencontre deux techniques dans la programmation dynamique :

- Technique récursive « Top Down » (de mémoïsation). Lors d'un appel récursif, on regarde dans une liste intermédiaire si le sous-problème est déjà traité.
  Top Down : on résout dans le sens des données de grande taille vers les données de petite taille.
- 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.

3. 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 de 0 inclus à n exclu.
- k varie de 0 inclus à n inclus.

Lorsque DIST$[i][j][k]$ n'a pas été calculé, 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)
```

4. 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)
```

5. 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 PRECEDENT$[i, j, 0]$:

- PRECEDENT$[i][j][0] = i$ si $i \neq j$ et $\mathbf{M}[i][j] < \infty$.

On utilise trois boucles `for` imbriquées pour calculer PRECEDENT$[i][j][k]$:

- PRECEDENT$[i][j][k]$ = PRECEDENT$[i][j][k-1]$ si $d_k(i, j) = d_{k-1}(i, j)$.
- PRECEDENT$[i][j][k]$ = PRECEDENT$[k-1][j][k-1]$ si

$$d_k(i, j) = d_{k-1}(i, k-1) + d_{k-1}(k-1, j) \text{ et } d_k(i, j) \neq \infty$$
$$\text{et PRECEDENT}[k-1][j][k-1] \neq -1$$

```
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)
```

Partie 12

Intelligence artificielle et jeux

Plan

15

Intelligence artificielle et jeu à deux joueurs (Spé)

Exercice 15.1 : Algorithme des *k* plus proches voisins

On considère un jeu de données « fic.csv » contenant *n* lignes. 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*). Les données sont séparées avec le séparateur « ; ». On utilise les listes de listes pour représenter les matrices dans Python.

L'instruction `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 l'objet fichier `f` la chaîne de caractères `'exemple'`

`f.close()` : ferme le fichier

1. Écrire une fonction `fic_data` qui admet comme argument un nom de fichier et retourne une matrice à *n* lignes et trois colonnes : longueur des pétales de l'iris, largeur des pétales de l'iris et désignation de l'espèce de l'iris (0, 1 ou 2).

2. 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 $(\text{longueurA}, \text{largeurA})$ et le point *B* de coordonnées $(\text{longueurB}, \text{largeurB})$.

3. 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 de l'iris (noté `iris2`) caractérisé par `pt_search` (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 `mat_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 `mat_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`.

4. Écrire le programme principal permettant d'afficher le nuage de points des données du fichier « fic.csv » ainsi que le point recherché `pt_search` et d'afficher une prédiction de l'espèce de `iris2` caractérisé par `pt_search=[5, 1.7]` avec $k = 5$.

Le graphique doit avoir les caractéristiques suivantes :

- affichage de « longueur des pétales » pour l'axe des abscisses et « largeur des pétales » pour l'axe des ordonnées ;
- affichage en bleu avec marqueur « v » pour les iris *setosa* ;
- affichage en rouge avec marqueur « . » pour les iris *versicolor* ;
- affichage en vert avec marqueur « + » pour les iris *virginica* ;
- affichage de la légende « iris setosa », « iris versicolor » et « iris virginica » ;
- affichage en noir avec `linewidth=8` pour l'iris `pt_search`.

On pourra se servir de la fonction `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()`.

Analyse du problème

En intelligence artificielle, l'algorithme des *k* plus proches voisins est une méthode d'apprentissage supervisé alors que l'algorithme des *k*-moyennes (voir exercice 15.3 « Algorithme des *k*-moyennes ») est une méthode d'apprentissage non supervisé.

L'objectif est de prédire la classe (ou la classification) d'un échantillon à partir d'exemples connus. On pourrait chercher le plus proche voisin de l'échantillon. L'inconvénient est que cette méthode du plus proche voisin est très sensible aux bruits. Une amélioration consiste à utiliser les k observations les plus proches. On cherche la classe majoritaire parmi les k plus proches voisins.

1.

```
def fic_data(fichier):
    # la fonction retourne une matrice :
    # n lignes et trois colonnes
    # 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
        # longueur, largeur 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 les caractères d'échappement
        # (ici saut de ligne)
        data[i][0], data[i][1]=float(ligne[0]), float(ligne[1])
        if (ligne[2]=="setosa"):
            data[i][2]=0     # numéro 0 désigne setosa
        elif (ligne[2]=="versicolor"):
            data[i][2]=1     # numéro 1 désigne versicolor
        else:                # virginica
            data[i][2]=2     # numéro 2 désigne virginica
    f.close()
    return data
```

2.

```
def calc_dist(ptA, ptB):
    # distance euclidienne entre ptA et ptB
    # ptA est une liste [longueurA, largeurA]
    # ptB est une liste [longueurB, largeurB]
    return ((ptB[0]-ptA[0])**2+(ptB[1]-ptA[1])**2)**(0.5)
```

Remarque :

On peut envisager plusieurs définitions de la distance entre $A\left(x_A, y_A\right)$ et $B\left(x_B, y_B\right)$:

- distance euclidienne : distance $= \sqrt{\left(x_B - x_A\right)^2 + \left(y_B - y_A\right)^2}$
- distance de Manhattan : distance $= \left|x_B - x_A\right| + \left|y_B - y_A\right|$
- distance de Tchebychev : distance $= \max\left(\left|x_B - x_A\right|, \left|y_B - y_A\right|\right)$

3. 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.

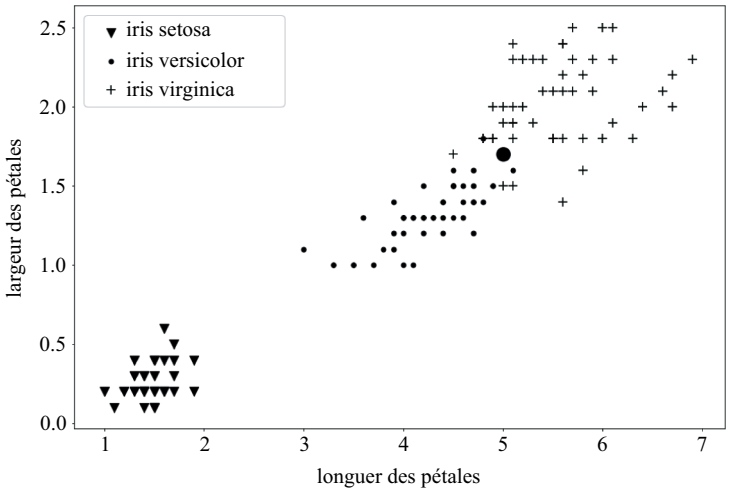
```
def algoknn(data, pt_search, k):
    # calcul de la distance de pt_search à tous les points
    # de data
    # la fonction retourne une prédiction de l'espèce de l'iris
    # algorithme des k(int) plus proches voisins
    # pt_search : liste de deux valeurs :
    # longueur et largeur des pétales
    n=len(data)                     # nombre de lignes de data
    tab_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)
        tab_dist[i][1]=data[i][2]
            # on récupère la distance et la désignation
            # de l'espèce
    tab_dist.sort()                 # tri par ordre croissant
                                    # des distances
    # on récupère la liste des distances triées par ordre
    # croissant
    # choix de l'espèce
    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(tab_dist[i][1]) # tab_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
```

4.

```
import matplotlib.pyplot as plt
    # module matplotlib.pyplot renommé plt
data=fic_data("fic.csv")
nb_espece=3          # on a trois espèces d'iris
    # setosa (0), versicolor (1) et virginica (2)
pt_search=[5, 1.7]   # longueur, largeur
k=5                  # nombre de k plus proches voisins
x1, y1, x2, y2, x3, y3=[],[],[],[],[],[]
for i in range(len(data)):
    if data[i][2]==0:
        x1.append(data[i][0])
        y1.append(data[i][1])
```

```
    elif data[i][2]==1:
        x2.append(data[i][0])
        y2.append(data[i][1])
    else:
        x3.append(data[i][0])
        y3.append(data[i][1])
plt.figure()
plt.xlabel("longueur des pétales")
plt.ylabel("largeur des pétales")
plt.scatter(x1, y1, color='blue', marker='v')
plt.scatter(x2, y2, color='red', marker='.')
plt.scatter(x3, y3, color='green', marker='+')
plt.scatter(pt_search[0], pt_search[1], color='black',\
            linewidth=8)
plt.legend(['iris setosa', 'iris versicolor',\
            'iris virginica'])
plt.show()

predict=algoknn(data, pt_search, k)
if predict==0:
    print(predict, 'setosa')
elif predict==1:
    print(predict, 'versicolor')
else:
    print(predict, 'virginica')
```

On obtient le graphique suivant avec le fichier « fic.csv » (voir le site Dunod pour télécharger le fichier de données).



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 `algoknn`, 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*.

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.

  Exemples : 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* = 3 classes (0 pour *setosa*, 1 pour *versicolor* et 2 pour *virginica*).
- **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.

Exercice 15.2 : Matrice de confusion, valeur optimale des *k* plus proches voisins

On considère les jeux de données « fic01.csv » (fichier d'apprentissage pour l'algorithme des *k* plus proches voisins) et « fic02.csv » (fichier de test). 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*). Les données sont séparées avec le séparateur « ; ». On utilise les listes de listes pour représenter les matrices dans Python. L'instruction `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 l'objet fichier `f` la chaîne de caractères `'exemple'`

`f.close()` : ferme le fichier

1. Écrire une fonction `fic_data` qui admet comme argument un nom de fichier et retourne une matrice à *n* lignes et trois colonnes : longueur des pétales de l'iris, largeur des pétales de l'iris et désignation de l'espèce de l'iris (0, 1 ou 2).

2. 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 d'un iris inconnu caractérisé par `pt_search` (liste de deux valeurs : longueur et largeur des pétales). On utilisera l'algorithme des *k* plus proches voisins.

3. 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 à 3 lignes et 3 colonnes :

		Classe réelle		
		setosa	versicolor	virginica
Classe prédite	setosa	…	…	…
	versicolor	…	…	…
	virginica	…	…	…

`mat[0][2]` représente le nombre de fois où l'algorithme prédit l'iris *setosa* alors qu'il est en réalité *virginica*.

4. On définit la matrice de confusion `matconf` pour l'iris *versicolor* avec $k = 2$:

		Classe réelle	
		versicolor	non versicolor
Classe prédite	versicolor	True Positives (TP)	False Positives (FP)
	non versicolor	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 ?

- $rappel = \dfrac{TP}{TP + FN}$

- $spécificité = \frac{TN}{TN+FP}$
- $précision = \frac{TP}{TP+FP}$

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 l'iris *versicolor* avec `k` = 2, `rappel`, `spécificité` et `précision`.

5. 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.

Analyse du problème

L'objectif de l'algorithme des *k* plus proches voisins est de prédire la classe (classification) ou la valeur (régression) d'un échantillon à partir d'exemples connus. On sépare les données en un jeu **d'entraînement** (fichier d'apprentissage : « fic01.csv ») et un **jeu de test** (fichier de test : « fic02.csv »). La répartition des données entre le jeu d'entraînement et le jeu de test peut être 80 %-20 %, 70 %-30 % ou même 50 %-50 %. Le jeu d'entraînement sert à apprendre le modèle (algorithme des *k* plus proches voisins). Le jeu de test sert à estimer l'erreur de généralisation du modèle.

1. Les deux premières questions reprennent les fonctions définies dans l'exercice précédent « Algorithme des *k* plus proches voisins ».

```
def fic_data(fichier):
    # la fonction retourne une matrice :
    # n lignes et trois colonnes
    # 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
        # longueur, largeur 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 les caractères d'échappement
        # (ici saut de ligne)
        data[i][0], data[i][1]=float(ligne[0]), float(ligne[1])
        if (ligne[2]=="setosa"):
            data[i][2]=0    # numéro 0 désigne setosa
        elif (ligne[2]=="versicolor"):
            data[i][2]=1    # numéro 1 désigne versicolor
        else:               # virginica
            data[i][2]=2    # numéro 2 désigne virginica
    f.close()
    return data
```

2.

```
def calc_dist(ptA, ptB):
    # distance euclidienne entre ptA et ptB
    # ptA est une liste [longueurA, largeurA]
    # ptB est une liste [longueurB, largeurB]
    return ((ptB[0]-ptA[0])**2+(ptB[1]-ptA[1])**2)**(0.5)

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 de l'iris
    # algorithme des k(int) plus proches voisins
    # pt_search : liste de deux valeurs :
    # longueur et largeur des pétales
    n=len(data)                     # nombre de lignes de data
    tab_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
        tab_dist[i][0]=calc_dist(pt_search,\
                                 [data[i][0],data[i][1]])
        tab_dist[i][1]=data[i][2]
            # on récupère la distance et la désignation
            # de l'espèce
    tab_dist.sort()                 # tri par ordre croissant
                                    # des distances
    # on récupère la liste des distances triées par ordre
    # croissant
    # choix de l'espèce
    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=tab_dist[i][1]       # tab_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
```

3.

```
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[predict][data2[i][2]]+=1
    return mat
```

4. On a plusieurs façons d'évaluer les données de test :

- Rappel = $\frac{TP}{TP + FN}$ = proportion des iris bien prédits (classe *versicolor*) parmi tous les iris *versicolor* dans le fichier de test. Le rappel définit la capacité du modèle à détecter la classe *versicolor* parmi les iris *versicolor* dans le jeu de test. On l'appelle également sensibilité.
- Spécificité = $\frac{TN}{TN + FP}$ = proportion des iris bien prédits (classe non-*versicolor*) parmi les iris qui ne sont pas de classe *versicolor* dans le fichier de test. La spécificité définit la capacité du modèle à détecter les iris qui ne sont pas de classe *versicolor* parmi les iris qui ne sont pas de classe *versicolor* dans le jeu de test.
- Précision = $\frac{TP}{TP + FP}$ = proportion des iris bien prédits (classe *versicolor*) parmi tous les iris dans le fichier de test. La précision définit la capacité du modèle à détecter la classe *versicolor* parmi toutes les classes dans le jeu de test.

True Positives (TP) :

		Classe réelle		
		setosa	versicolor	virginica
Classe prédite	setosa	…	…	…
	versicolor	…	…	…
	virginica	…	…	…

True Negatives (TN) :

		Classe réelle		
		setosa	versicolor	virginica
Classe prédite	setosa	…	…	…
	versicolor	…	…	…
	virginica	…	…	…

False Negatives (FN) :

		Classe réelle		
		setosa	versicolor	virginica
Classe prédite	setosa	…	…	…
	versicolor	…	…	…
	virginica	…	…	…

False Positives (FP) :

		Classe réelle		
		setosa	versicolor	virginica
Classe prédite	setosa	…	…	…
	versicolor	…	…	…
	virginica	…	…	…

```
def matconf_indicateurs (data1, data2):
    # data1 : liste d'apprentissage ; data2 : liste de test
    # la fonction retourne la matrice de confusion pour
    # l'iris versicolor (k = 2)
    # 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]+mat[0][2]+mat[2][0]+mat[2][2]
                                      # True Negatives (TN)
    matconf[1][0]=mat[0][1]+mat[2][1] # False Negatives (FN)
    matconf[0][1]=mat[1][0]+mat[1][2] # 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
```

Remarque : On peut avoir des faux négatifs ou des faux positifs avec les tests pour la Covid.

5. On sépare le fichier de l'exercice précédent « Algorithme des *k* plus proches voisins » en deux fichiers : `fichier1` (fichier d'apprentissage correspondant au jeu d'entraînement) et `fichier2` (fichier de test correspondant au jeu de test).

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,tab_pred[1][1]
```

On applique les fonctions avec les fichiers « fic01.csv » et « fic02.csv » (voir le site Dunod pour télécharger les fichiers de données).

```
tab_pred, val_k_optimisee=predict_k(data1, data2)
print("Valeur optimisée pour k = ", val_k_optimisee)
```

Le programme Python affiche : `Valeur optimisée pour k=6`.

Exercice 15.3 : 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. On utilise les listes de listes pour représenter les matrices dans Python.

On pourra se servir de la fonction `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()`.

Rappels pour la génération de nombres aléatoires :

```
import random as rd  # module random renommé rd
rd.random()          # nombre flottant aléatoire M tel que 0 <= M < 1
rd.randint(a, b)     # renvoie un entier aléatoire M tel que a <= M <= b
```

On définit la variable `inf=1e10` qui représente une distance infinie.

1. On étudie dans cette question un nuage de 12 points que l'on représentera sur feuille afin de comprendre le début de l'algorithme itératif.

On considère deux listes `X` et `Y` représentant les abscisses et les ordonnées de 3 groupes de 4 points générés dont les coordonnées sont comprises dans les intervalles $[3\pm0.6,1\pm0.6]$, $[8\pm1,2\pm1]$ et $[4\pm0.8,5\pm0.8]$ pour chaque groupe. Représenter sur feuille un nuage de points.

- Initialisation des centres des clusters : On choisit aléatoirement 3 points parmi les 12 points. Représenter sur feuille le nuage de points en mettant en évidence les 3 clusters.
- On réalise une première partition des données en associant chacun des autres points au cluster le plus proche. On calcule alors les centres des nouveaux clusters. Représenter sur feuille le nuage de points en mettant en évidence les 3 nouveaux clusters.

2. Écrire une fonction `initpoints` qui admet comme argument un entier `pts_groupe` et retourne deux listes `X` et `Y` contenant chacune `N` = 3×`pts_groupe` valeurs. Les listes *X* et *Y* représentent les coordonnées des *N* points. Chaque groupe est constitué de `pts_groupe` points. Les 3 groupes contiennent des points de coordonnées comprises dans les intervalles $[3\pm2.6,1\pm2.6]$, $[8\pm2.5,2\pm2.5]$ et $[4\pm2.8,5\pm2.8]$.

3. Écrire une fonction `distance` 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 $\left(\text{ptB}[0], \text{ptB}[1]\right)$.

4. Écrire une fonction `initcluster` qui 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.

5.

Algorithme des k-moyennes

Initialisation :

On choisit au hasard *K* centres des clusters parmi le nuage de points. Chaque centre est caractérisé par une abscisse et une ordonnée.

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.

Écrire une fonction `algokm` qui 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` telle que `A[i]` désigne le numéro du cluster du point `[X[i],Y[i]]`. Quels sont les défauts de cet algorithme ?

6. Écrire le programme principal permettant :

- de générer aléatoirement deux listes `X` et `Y` représentant les abscisses et les ordonnées de 3 groupes de 100 points dont les coordonnées sont comprises dans les intervalles $[3\pm2.6,1\pm2.6]$, $[8\pm2.5,2\pm2.5]$ et $[4\pm2.8,5\pm2.8]$ pour chaque groupe ;
- d'afficher graphiquement le nuage de points (affecter des couleurs différentes pour les points appartenant à des clusters différents) et les centres des 3 clusters.

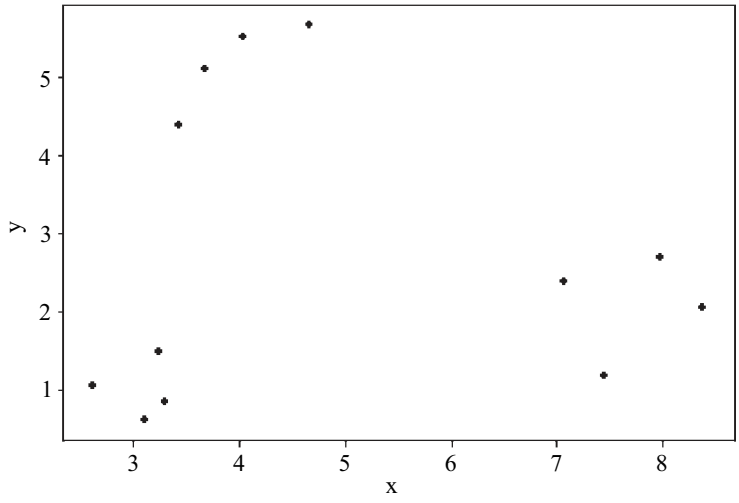
7. Écrire une fonction `predict` d'arguments d'entrée `L` (liste de listes contenant les coordonnées des clusters) et une liste `pt_search` (abscisse et ordonnée du point). Cette fonction retourne le numéro et les coordonnées du cluster le plus proche de `pt_search`.

Analyse du problème

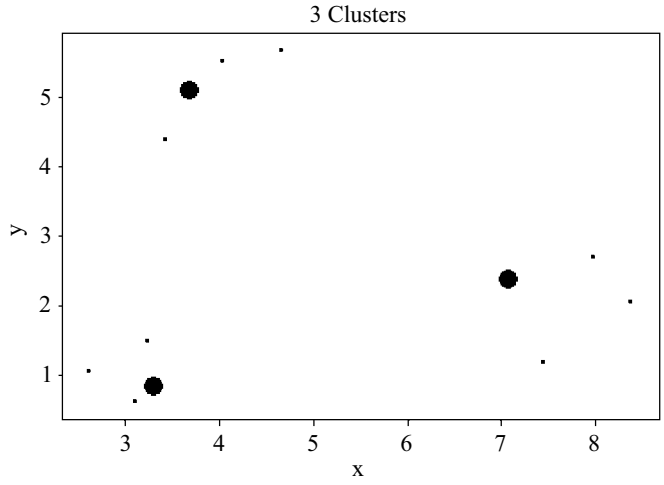
Dans l'apprentissage non supervisé, l'objectif est de comprendre la structure des données. Contrairement à l'apprentissage supervisé (voir exercice 15.1 « Algorithme des *k* plus proches voisins » avec la prédiction de la classe d'un échantillon à partir d'exemples connus), on ne connaît pas les clusters. En pratique, on regroupe les données proches entre elles, on leur attribue des clusters qu'il faut interpréter ensuite.

Exemple : regrouper des clients qui ont des profils similaires, regrouper les documents d'un corpus par thème (les thèmes émergeant de cet algorithme ne sont pas connus). C'est une méthode d'apprentissage non supervisé puisqu'on ne connaît pas les clusters à l'avance.

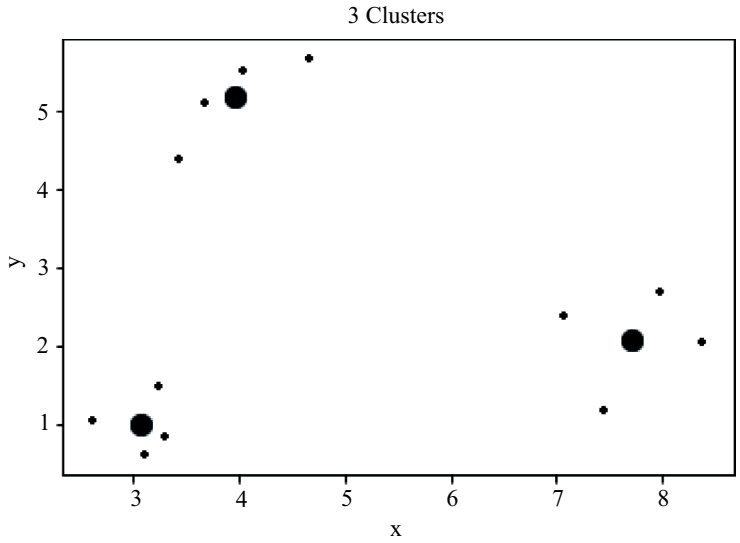
1. On considère un nuage de 12 points.



- Étape d'initialisation : on choisit aléatoirement 3 centres parmi le nuage de 12 points.



- Première partition : on affecte à chaque point du jeu de données le cluster le plus proche. On met à jour les nouveaux centres des clusters. On obtient trois nouveaux clusters représentés ci-dessous.



Remarque : Les nouveaux clusters n'appartiennent plus nécessairement au jeu de données comme dans l'étape d'initialisation. On affecte à chaque point le cluster le plus proche. On met à jour les centres des clusters. Ces opérations sont réitérées tant que les points changent de cluster.

2.

```
import random as rd          # module random renommé rd

def initpoints(pts_groupe):
    # la fonction retourne deux listes X et Y
    # contenant 3 groupes
    # le nombre de valeurs aléatoire pour chaque groupe
    # est pts_groupe
    N=pts_groupe*3 #   N = nombre total de points
    X=[0 for i in range(N)] # liste des abscisses pour
                            # les N points
    Y=[0 for i in range(N)] # liste des ordonnées pour
                            # les N points
    for k in range(3):      # k varie entre 0 inclus et 3 exclu
        if k==0:
            xcentre=3       # abscisse du centre du premier
                            # groupe (ex.3)
            ycentre=1       # ordonnée du centre du premier
                            # groupe (ex. 1)
            delta=2.6       # 2.6 ou 0.6
        elif k==1:
            xcentre=8       # abscisse du centre du deuxième
                            # groupe (ex. 8)
            ycentre=2       # ordonnée du centre du premier
                            # groupe (ex. 2)
            delta=2.5       # 2.5 ou 1
        else:
            xcentre=4       # abscisse du centre du troisième
                            # groupe (ex. 4)
            ycentre=5       # ordonnée du centre du troisième
                            # groupe (ex. 5)
            delta=2.8       # 2.8 ou 0.8
        for i in range(pts_groupe):  # i varie entre 0 inclus
                                     # et pts_groupe exclu
            x=xcentre+delta*(2*rd.random()-1)
                # x compris entre xcentre-delta et xcentre+delta
            y=ycentre+delta*(2*rd.random()-1)
                # y compris entre ycentre-delta et ycentre+delta
            X[i+k*pts_groupe]=x
            Y[i+k*pts_groupe]=y
    return X, Y
```

3.

```
def distance(ptA, ptB):
    # distance euclidienne entre ptA et ptB
    # ptA est une liste [xA, yA] désignant l'abscisse
    # et l'ordonnée de A
    # ptB est une liste [xB, yB] désignant l'abscisse
    # et l'ordonnée de B
    return ((ptB[0]-ptA[0])**2+(ptB[1]-ptA[1])**2)**(0.5)
```

4. On choisit aléatoirement *K* points différents parmi le nuage de points. 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, 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
```

Remarque : Les points choisis aléatoirement dans cette étape d'initialisation doivent appartenir au nuage de points.

5.

```
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
```

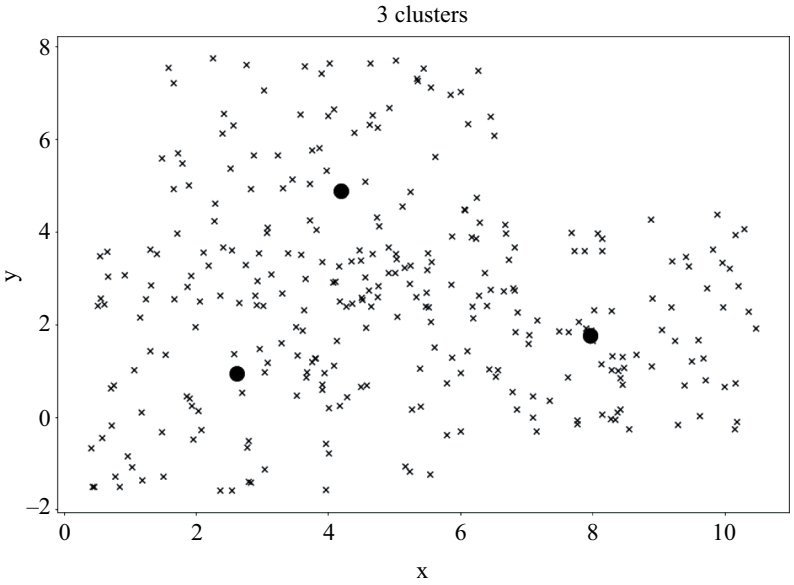
Cet algorithme a plusieurs défauts :

- 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 optimum.
- On peut obtenir un minimum local qui dépend des centres initiaux.

6.

```
import matplotlib.pyplot as plt
    # module matplotlib.pyplot renommé plt
pts_groupe=100                  # nombre de points pour chaque
                                # groupe
N=pts_groupe*3                  # nombre total de points
X, Y=initpoints(pts_groupe)
inf=1e10                        # variable représentant
                                # l'infini
K=3
L3, A=algokm (K, X, Y)
plt.figure()                    # nouvelle fenêtre graphique
```

```
plt.scatter(X,Y,color='black')
plt.scatter(L3[0][0],L3[0][1],color='red')
plt.scatter(L3[1][0],L3[1][1],color='blue')
plt.scatter(L3[2][0],L3[2][1],color='green')
plt.xlabel('x')
plt.ylabel('y')
plt.title('3 clusters')
plt.show()                          # affiche la figure à l'écran
print(L3)                           # affichage des coordonnées
                                    # des clusters
```



Les coordonnées des 3 clusters sont : [[7.96, 1.76], [2.62, 0.94], [4.20, 4.87]].

Les centres correspondent bien aux 3 groupes centrés sur [[8, 2], [3, 1], [4, 5]].

Remarque : En pratique, on utilise cet algorithme pour former des groupes inconnus à l'avance mais qu'il faut interpréter ensuite.

7.

```
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
    # centres des clusters, pt_search = liste contenant
    # l'abscisse et l'ordonnée de pt_search
    # L[i] = liste contenant l'abscisse et l'ordonnée
    # du cluster d'indice i
    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]
```

Exercice 15.4 : 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	.

1. 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.

2. 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`.

3. Principe de l'algorithme min-max :

- Écrire une fonction `valMax` qui admet comme argument une 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 une 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.

4. 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.

5. 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.

Analyse du problème

L'algorithme min-max est un algorithme très utilisé dans les jeux à somme nulle et à nombre fini de stratégies. On explore toutes les possibilités. On définit un gain pour chaque joueur. À chaque coup, les joueurs cherchent à maximiser leur gain minimum et donc à minimiser le gain maximum de l'adversaire.

Cours

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) et 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.

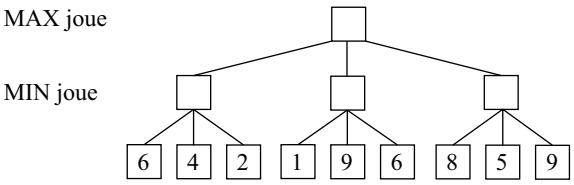
L'algorithme min-max est utilisé dans de nombreux jeux : Othello, échecs…

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.

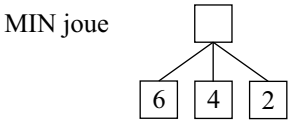
Les feuilles sont les nœuds terminaux pour lesquels ne partent aucune branche et correspondent souvent à une fin de partie. La hauteur de l'arbre ci-dessus est 3. Pour les jeux, on parle plutôt de profondeur de jeu que de hauteur. La profondeur pouvant être très importante pour arriver à une fin de partie, on limite souvent la profondeur d'étude (voir exercice suivant « Jeu de morpion, algorithme min-max et profondeur »). 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.

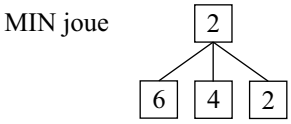
Le joueur MAX a trois possibilités pour jouer et choisit le coup qui va maximiser ses gains. Lorsque le joueur MAX a joué, le joueur MIN va jouer en cherchant à minimiser ses gains.

On parcourt en profondeur cet arbre en utilisant la fonction GAIN pour déterminer le meilleur coup à jouer pour MAX.

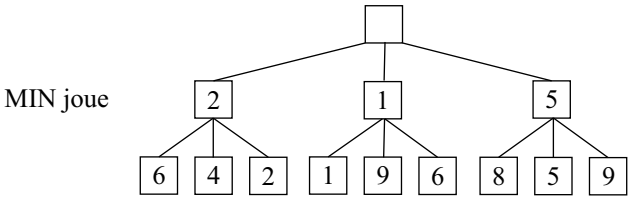
On considère les feuilles : 6, 4 et 2. On remonte dans l'arbre. C'est à MIN de jouer.



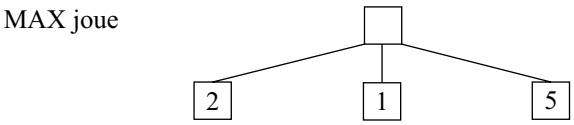
Il choisit le coup qui minimise le gain. Il choisit alors le coup avec un gain égal à 2.



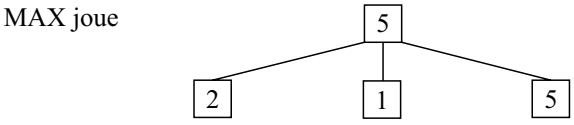
On complète l'arbre de jeu avec le minimum des gains.



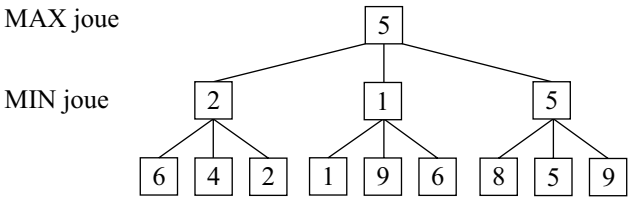
On remonte dans l'arbre et on se pose la question : Quel coup choisir pour le joueur1 (MAX) ?



Le joueur1 (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.

1.

```
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):
    # la fonction 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):
    # la fonction 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
```

2. 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 exercice 1.4 « Affectation, objet immuable, copie » dans le chapitre « Prise en main de Python ») pour réaliser une copie superficielle de `jeu`.

```
def jouercoup(jeu, coup):
    # la fonction 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
```

3.

```
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
```

4.

```
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)
```

5.

```
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 ; 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)
```

Exercice 15.5 : Jeu de morpion, algorithme min-max et profondeur

Cet exercice est la suite de l'exercice précédent « Jeu de morpion, algorithme min-max ». On pourra utiliser les fonctions `init`, `affiche`, `choixjoueur`, `listecoups`, `gain` et `jouercoup`.

1. 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.

2. Écrire une fonction `jouercontreIA2` 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.

Analyse du problème

L'algorithme min-max peut nécessiter un temps de calcul très important pour parcourir toutes les branches en profondeur. Dans cet exercice, on limite la profondeur d'étude dans l'arbre de jeu. On peut ainsi déterminer un niveau de difficulté du jeu.

1.

```
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
```

2.

```
def jouercontreIA2(jeu, profondeurmaxi):
    # cette fonction affiche les plateaux de jeu à chaque étape
    # la liste jeu est modifiée à chaque étape
    # ordinateur ou IA : pions noirs ; l'humain : pions blancs
    # profondeurmaxi = nombre de coups calculés à l'avance par
    # l'algorithme
    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=valMax2(jeu, profondeurmaxi)
            # 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 : "))
            # 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 en choisissant le niveau de difficulté et la couleur des pions est :

```
jeu=init()          # début de partie avec cases '.'
profondeurmaxi=10   # 2 : niveau débutant, 6 : intermédiaire,
                    # 10 : expert
jouercontreIA2(jeu, profondeurmaxi)
```

Partie 13

Bases de données

Plan

Bases de données (Spé)

16

Exercice 16.1 : Joueurs de tennis

On considère la base de données `TENNIS` pour gérer les joueurs de tennis.
La table `joueurs` contient les colonnes :

- `id_joueur`, de type entier, identifie chaque joueur ;
- `nom`, de type chaîne de caractères, désigne le nom du joueur ;
- `annee`, de type entier, désigne l'année de naissance ;
- `nationalite`, de type chaîne de caractères, désigne la nationalité du joueur.

1	MURRAY	1987	britannique
2	GULBIS	1988	letton
3	FEDERER	1981	suisse
4	DJOKOVIC	1987	serbe
5	BERDYCH	1985	tchèque
6	NADAL	1986	espagnole
7	THIEM	1993	autrichienne
8	NISHIKORI	1989	japonaise
9	TSONGA	1985	française
10	WAWRINKA	1985	suisse
11	MONFILS	1986	française
12	SIMON	1984	française

1. Écrire une requête SQL qui renvoie toutes les informations de tous les joueurs.

2. Écrire une requête qui renvoie le nom de tous les joueurs français.

3. Écrire une requête qui renvoie la liste des nationalités des joueurs de tennis.

4. Écrire une requête qui renvoie la moyenne des années de naissance des joueurs français.

5. Écrire une requête qui renvoie la nationalité, l'année de naissance et le nom des joueurs dont l'année de naissance est supérieure ou égale à 1988. On renommera les colonnes nationalité et année.

6. Écrire une requête qui renvoie l'année de naissance et le nom des joueurs dont l'année de naissance est strictement supérieure à 1980 et de nationalité suisse.

7. Écrire une requête qui renvoie le nombre de joueurs ayant la même année de naissance. Les années sont affichées par ordre croissant.

8. Écrire une requête qui renvoie les identifiants et les noms des 3 premiers joueurs sans le tout premier de la table.

Analyse du problème

Dans cet exercice, on n'utilise qu'une seule table. Pour lire les données d'une base de données, on utilise la commande `SELECT` qui retourne les enregistrements sélectionnés dans un tableau.

Cours :

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 de blanc pour désigner les colonnes (que l'on peut appeler attributs). Le type des colonnes peut être entier (INTEGER), flottant (FLOAT) ou chaîne (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`, `NOT` dans la condition `WHERE`.

Les opérateurs de comparaison sont :

>	<	>=	<=	=	<>
Strictement supérieur à	Strictement inférieur à	Supérieur ou égal à	Inférieur ou égal à	Égal à	Différent de

```
SELECT *
FROM table
WHERE condition1 AND condition2;
```

On peut ajouter `DISTINCT` après `SELECT` pour éviter d'afficher des lignes en double.

```
SELECT DISTINCT colonne
FROM table ;
```

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;
```

On peut utiliser d'autres fonctions statistiques : `MAX` (maximum), `MIN` (minimum), `SUM` (somme), `AVG` (moyenne).

`AS` permet de renommer une colonne ou une table. On utilisera la requête suivante :

```
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`.

La commande `LIMIT 3` permet de sélectionner les trois premiers résultats.

```
SELECT colonne1, colonne2
FROM table
LIMIT 3;
```

La commande `LIMIT 3` permet de sélectionner les trois premiers résultats sans utiliser les deux premiers de `table`.

```
SELECT colonne1, colonne2
FROM table
LIMIT 3 OFFSET 2;
```

Voir exercice 16.4 « Imprimantes (Mines Ponts 2015) » pour une requête imbriquée.

Remarque : On pourra tester les requêtes SQL en utilisant SQLite Database Browser (https://sqlitebrowser.org).

1.

```
SELECT * FROM joueurs;
```

`*` permet de retourner toutes les colonnes.

2.

```
SELECT nom
FROM joueurs
WHERE nationalite='française';
```

On obtient alors : TSONGA, MONFILS, SIMON.

3. On ajoute `DISTINCT` pour éviter d'afficher plusieurs fois la même nationalité.

```
SELECT DISTINCT nationalite
FROM joueurs;
```

Cours :
Les fonctions d'agrégation permettent de réaliser des opérations statistiques.

```
SELECT COUNT(*)
FROM joueurs;
```

La commande `COUNT(*)` compte alors le nombre de lignes. On obtient : 12.

On utilise souvent les fonctions statistiques : `MAX` (maximum), `MIN` (minimum), `SUM` (somme) et `AVG` (moyenne).

```
SELECT MAX(annee)
FROM joueurs;
```

On obtient : 1993.

4.

```
SELECT AVG(annee)
FROM joueurs
WHERE nationalite='française';
```

5.

```
SELECT nationalite AS 'nationalité', annee AS 'année', nom
FROM joueurs
WHERE annee >=1988;
```

On obtient alors : 1988 GULBIS, 1989 NISHIKORI et 1993 THIEM.

6.

```
SELECT annee, nom
FROM joueurs
WHERE annee >1980 AND nationalite='suisse';
```

Remarque :

On peut écrire également :

```
SELECT annee, nom
FROM joueurs
WHERE (annee >1980 AND nationalite='suisse');
```

Cours :
La commande `GROUP BY` évite d'avoir plusieurs fois la même ligne. On regroupe les lignes d'un même joueur en une seule.

La commande `COUNT(*)` compte alors le nombre de lignes concernées.

La commande `ORDER BY annee` permet de trier les lignes par année. Par défaut, le tri est par ordre croissant (ou ascendant).

Les fonctions d'agrégation `SUM(nom_col)`, `AVG(nom_col)`, `MAX(nom_col)`, `MIN(nom_col)`, `COUNT(nom_col)`, `COUNT(*)` calculent respectivement la somme, la moyenne arithmétique, le maximum, le minimum, le nombre de valeurs non nulles de la colonne `nom_col` 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.

7.

```
SELECT annee, COUNT(*)
FROM joueurs
GROUP BY annee
ORDER BY annee;
```

8.

```
SELECT id_joueur, nom
FROM joueurs
LIMIT 3 OFFSET 1;
```

La commande `LIMIT 3 OFFSET 1` permet d'afficher les trois premiers résultats sans utiliser le premier résultat de la table `joueurs`. On obtient les lignes 2, 3 et 4.

Exercice 16.2 : Tournois et joueurs de tennis

On reprend la base de données `TENNIS` définie dans l'exercice précédent, mais qu'on appellera ici `joueurs`. On ajoute une autre table `tournois` qui contient les colonnes :

- `id_tournoi`, de type entier, identifie chaque tournoi ;
- `nom`, de type chaîne de caractères, désigne le nom du tournoi ;
- `annee`, de type entier, désigne l'année où a lieu le tournoi ;
- `num_finaliste1`, de type entier, identifie le vainqueur du tournoi ;
- `num_finaliste2`, de type entier, identifie le perdant de la finale du tournoi ;
- `num_joueur3`, de type entier, identifie le troisième joueur en demi-finale ;
- `num_joueur4`, de type entier, identifie le quatrième joueur en demi-finale ;
- `gain`, de type entier, désigne la somme gagnée par le vainqueur.

1	ROLAND-GARROS	2016	4	1	7	10	2000000
2	ROLAND-GARROS	2015	10	4	9	1	1800000
3	ROLAND-GARROS	2014	6	4	1	2	1650000
4	OPEN AUSTRALIE	2016	4	8	9	12	2390000
5	OPEN AUSTRALIE	2015	4	1	5	10	2100000
6	OPEN AUSTRALIE	2014	10	6	3	5	2050000

1. Qu'appelle-t-on une clé primaire ? Qu'appelle-t-on une clé étrangère ?

2. Écrire une requête SQL qui renvoie le nom du vainqueur de Roland-Garros en 2016.

3. Écrire une requête qui renvoie pour chaque joueur l'année et le nom des tournois gagnés par celui-ci. Les noms des joueurs sont triés par ordre croissant.

4. Écrire une requête qui renvoie le nombre de victoires pour chaque joueur.

5. Écrire une requête qui renvoie le total des gains pour chaque joueur.

6. Écrire une requête qui renvoie le gain moyen de chaque joueur d'origine serbe.

7. Écrire une requête qui renvoie le gain moyen de chaque tournoi. Les noms des tournois sont triés par ordre croissant.

8. Écrire une requête qui renvoie la liste des joueurs ayant gagné au moins deux tournois et nés après 1986. On affichera également le nombre de tournois gagnés.

9. Écrire une requête qui renvoie le joueur ayant gagné le plus de tournois.

10. Écrire une requête qui renvoie pour chaque joueur le nombre de participations à une demi-finale d'un tournoi. On renommera les tables `joueurs` et `tournois` respectivement `j` et `t`. Les joueurs sont affichés par ordre croissant.

11. Représenter le schéma relationnel de la base de données.

Analyse du problème

On utilise dans cet exercice deux tables : `joueurs` et `tournois`. On cherche à les mettre en relation en utilisant plusieurs clés (joueurs en demi-finale). On pourra utiliser `JOIN… ON…` pour réaliser une jointure. La commande `HAVING` permet de réaliser des fonctions d'agrégation comme `COUNT`.

Cours :

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 joueurs
JOIN tournois;
```

Cette requête réalise le produit cartésien des deux tables `joueurs` et `tournois`. À chaque ligne de la table `tournois`, il accole l'ensemble des lignes de la table `joueurs`. Le nombre de lignes affichées vaut 12×6=72. On a réalisé une jointure entre les deux tables.

id_joueur	nom	annee	nationalite	id_tournoi	nom	annee	(1)	(2)	(3)	(4)	gain
1	MURRAY	1987	britannique	1	ROLAND-GARROS	2016	4	1	7	10	2000000
1	MURRAY	1987	britannique	2	ROLAND-GARROS	2015	10	4	9	1	1800000

(1) `num_finaliste1` ; (2) `num_finaliste2` ; (3) `num_joueur3` ; (4) `num_joueur4`

On peut préciser une condition de jointure avec le mot-clé `ON`.

```
SELECT *
FROM joueurs
JOIN tournois ON id_joueur=num_finaliste1;
```

La requête SQL n'affiche pas les 72 lignes mais uniquement celles dont la condition `id_joueur=num_finaliste1` est vérifiée. On a réalisé une jointure interne. On aurait pu écrire `INNER JOIN` au lieu de `JOIN`.

`GROUP BY` permet de regrouper les lignes en une seule. `HAVING` fait quasiment la même chose que `WHERE` mais permet d'utiliser des fonctions d'agrégation comme `COUNT` pour compter le nombre d'éléments.

La requête est alors la suivante :

```
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;
```

On rencontre parfois des jointures d'une table sur elle-même. On parle d'autojointure (voir exercice 16.3 « Numéros de sécurité sociale »).

1. Une clé primaire sert à identifier une ligne de manière unique. Chaque joueur est désigné par un numéro d'identifiant `id_joueur`. Chaque tournoi est désigné par un numéro d'identifiant `id_tournoi`.

Une clé étrangère permet de lier des relations (ou tables) entre elles. La clé `num_finaliste1` permet d'avoir le numéro d'identifiant du joueur qui a gagné la finale du tournoi. Les clés `num_finaliste1`, `num_finaliste2`, `num_joueur3` et `num_joueur4` sont des clés étrangères.

2. Il faut réaliser une jointure entre les tables `tournois` et `joueurs`. La colonne `nom` peut prêter à confusion puisqu'elle est utilisée dans les deux tables. Le nom du joueur est alors désigné par `joueurs.nom`.

```
SELECT joueurs.nom
FROM joueurs
JOIN tournois ON num_finaliste1=id_joueur
WHERE tournois.nom='ROLAND-GARROS' AND tournois.annee=2016;
```

On obtient alors : DJOKOVIC.

Remarque :

On peut écrire également :

```
SELECT joueurs.nom
FROM joueurs, tournois
WHERE tournois.nom='ROLAND-GARROS' AND tournois.annee=2016
AND num_finaliste1=id_joueur;
```

Par la suite, on utilisera la commande `JOIN… ON…`

3.

```
SELECT joueurs.nom, tournois.nom, tournois.annee
FROM joueurs
JOIN tournois ON num_finaliste1=id_joueur
ORDER BY joueurs.nom;
```

On obtient alors : DJOKOVIC Roland-Garros 2016, DJOKOVIC Open d'Australie 2061, DJOKOVIC Open d'Australie 2015, NADAL Roland-Garros 2014, WAWRINKA Roland-Garros 2015, WAWRINKA Open d'Australie 2014.

Cours :
On peut ajouter `ASC` pour préciser que le tri se fait par ordre croissant. Ce n'est pas obligatoire puisque le tri se fait par défaut par ordre croissant.

`ORDER BY joueurs.nom ASC` : tri des noms par ordre croissant.

`ORDER BY joueurs.nom DESC` : tri des noms par ordre décroissant.

Cours :
```
SELECT AVG(gain)
FROM tournois;
```

On obtient : 1998333.33.

On peut utiliser des fonctions d'agrégation avec la commande `GROUP BY`.

```
SELECT nom, AVG(gain)
FROM tournois
GROUP BY nom;
```

La commande `GROUP BY` évite d'avoir plusieurs fois la même ligne. Il faut donc regrouper les lignes d'un même tournoi.

La commande `AVG(gain)` donne la moyenne des gains pour chaque tournoi.

On obtient :

OPEN AUSTRALIE	2180000.0
ROLAND-GARROS	1816666.67

4.
```
SELECT joueurs.nom, COUNT(*)
FROM joueurs, tournois
WHERE num_finaliste1=id_joueur
GROUP BY joueurs.nom;
```

On regroupe les lignes d'un même joueur en une seule. La commande `COUNT(*)` compte alors le nombre de lignes concernées.

On obtient alors : DJOKOVIC 3, NADAL 1, WAWRINKA 2.

5.
```
SELECT joueurs.nom, SUM(tournois.gain)
FROM joueurs, tournois
WHERE num_finaliste1=id_joueur
GROUP BY joueurs.nom;
```

On obtient alors : DJOKOVIC 6490000, NADAL 1650000, WAWRINKA 3850000.

6.
```
SELECT joueurs.nom, AVG(tournois.gain)
FROM joueurs, tournois
WHERE num_finaliste1=id_joueur AND nationalite='serbe'
GROUP BY joueurs.nom;
```

7.

```
SELECT nom, AVG(tournois.gain)
FROM tournois
GROUP BY nom
ORDER BY nom;
```

8.

```
SELECT joueurs.nom, COUNT(joueurs.nom)
FROM joueurs
JOIN tournois ON num_finaliste1=id_joueur
WHERE joueurs.annee>1986
GROUP BY joueurs.nom
HAVING COUNT(joueurs.nom)>=2;
```

9.

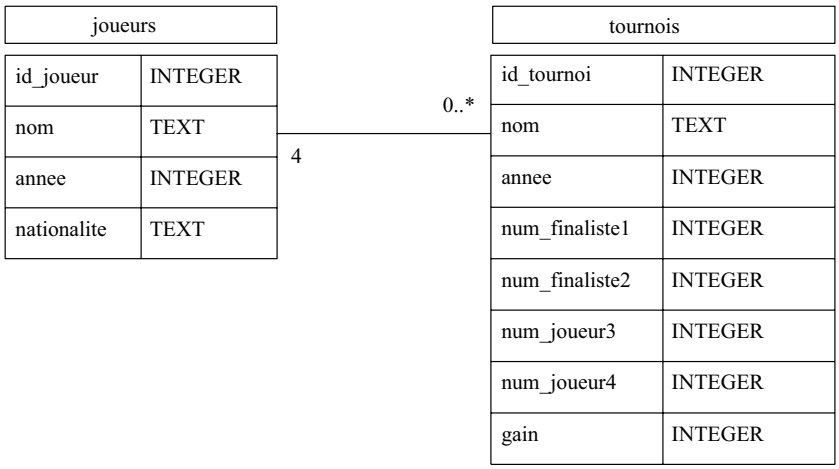
```
SELECT joueurs.nom, COUNT(joueurs.nom)
FROM joueurs
JOIN tournois ON num_finaliste1=id_joueur
GROUP BY joueurs.nom
ORDER BY COUNT(joueurs.nom) DESC
LIMIT 1;
```

Cours :
`LIMIT 1` permet d'afficher le premier résultat.

10.

```
SELECT j.nom, COUNT(j.nom)
FROM joueurs AS j
JOIN tournois as t ON num_finaliste1=id_joueur OR num_
finaliste2=id_joueur OR num_joueur3=id_joueur OR num_joueur4=id_
joueur
GROUP BY j.nom
ORDER BY j.nom;
```

11.



> Un joueur peut être lié à aucun tournoi ou à plusieurs tournois. On écrit alors la multiplicité `0..*`.
> Un tournoi est lié à exactement quatre joueurs. On écrit alors la multiplicité `4..4`, que l'on note `4`.

Cours :
Une association est un lien entre deux ou plusieurs entités.

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

Exercice 16.3 : 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

1. 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.

2. 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.

3. Écrire une requête qui renvoie le nom et le prénom des assurés qui ont les mêmes noms et prénoms en Bretagne et Nouvelle-Aquitaine.

4. 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.

5. 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.

6. 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.

Analyse du problème

On utilise dans cet exercice des opérateurs ensemblistes : `UNION`, `INTERSECT` et `EXCEPT`. Ils permettent de combiner dans un résultat unique des lignes provenant de deux requêtes `SELECT`.

Cours :

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

1.

```
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;
```

Cours :
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

2.

```
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

Cours :
La commande `INTERSECT` permet d'obtenir l'intersection de deux requêtes `SELECT`, c'est-à-dire les enregistrements communs aux deux requêtes.

3.

```
SELECT nom, prenom FROM tab_bretagne
INTERSECT
SELECT nom, prenom FROM tab_aquitaine;
```

Cours :
La commande `EXCEPT` permet de récupérer les enregistrements de la première requête `SELECT` sans inclure les résultats de la deuxième requête `SELECT`.

4.

```
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

5.

```
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

6.

```
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

Exercice 16.4 : Imprimantes (Mines Ponts 2015)

Une représentation simplifiée de deux tables de la base de données `imprimantes` est donnée ci-dessous :

- table `testfin` :

nSerie	dateTest	…	Imoy	Iec	…	fichierMes
230-588ZX2547	2012-04-22 14-25-45		0.45	0.11		mesure31025.csv
230-588ZX2548	2012-04-22 14-26-57		0.43	0.12		mesure41026.csv

- table `production` :

Num	nSerie	dateProd	type
20	230-588ZX2547	2012-04-22 15-52-12	JETDESK-1050
21	230-588ZX2549	2012-04-22 15-53-24	JETDESK-3050

Après son assemblage et avant les différents tests de validation, un numéro de série unique est attribué à chaque imprimante. À la fin des tests de chaque imprimante, les résultats d'analyse ainsi que le fichier contenant l'ensemble des mesures réalisées sur l'imprimante sont rangés dans la table `testfin`. Lorsqu'une imprimante satisfait les critères de validation, elle est enregistrée dans la table `production` avec son numéro de série, la date et l'heure de sortie de production ainsi que son type.

1. Écrire une requête SQL permettant d'obtenir les numéros de série des imprimantes ayant une valeur de `Imoy` comprise strictement entre deux bornes `Imin` et `Imax`.

2. Écrire une requête permettant d'obtenir les numéros de série, la valeur de l'écart type (`Iec`) et le fichier de mesures des imprimantes ayant une valeur de `Iec` strictement inférieure à la valeur moyenne de la colonne `Iec`.

3. Écrire une requête permettant d'extraire à partir de la table `testfin` le numéro de série et le fichier de mesures correspondant aux imprimantes qui n'ont pas été validées en sortie de production.

Analyse du problème

Cet exercice est extrait de l'épreuve d'informatique du concours Mines Ponts 2015.

On utilise une rubrique imbriquée. Le résultat d'une requête imbriquée en SQL peut retourner un champ (question 2) ou une colonne (question 3).

1.

```
SELECT nSerie
FROM testfin
WHERE Imoy>Imin and Imoy<Imax;
```

2.

```
SELECT nSerie,Iec,fichierMes
FROM testfin
WHERE Iec<(SELECT AVG(Iec) FROM testfin);
```

Cours :

La requête imbriquée (`SELECT AVG(Iec) FROM testfin`) retourne un champ : la valeur moyenne de la colonne `Iec`.

3.

```
SELECT nSerie,fichierMes
FROM testfin
WHERE nSerie NOT IN (SELECT nSerie from production);
```

La requête imbriquée (`SELECT nSerie from production`) retourne une colonne : les numéros de série de la table `production`.

Remarque :

La requête suivante permet de tester si le numéro de série de la table `testfin` est également dans la table `production`.

```
SELECT nSerie, fichierMes
FROM testfin
WHERE nSerie IN (SELECT nSerie from production);
```

Exercice 16.5 : Paludisme (Mines Ponts 2016)

Pour suivre la propagation des épidémies, de nombreuses données sont recueillies par les institutions internationales comme l'O.M.S. Par exemple, pour le paludisme, on dispose de deux tables :

- La table `palu` recense le nombre de nouveaux cas confirmés et le nombre de décès liés au paludisme ; certaines lignes de cette table sont données en exemple (on précise que `iso` est un identifiant unique pour chaque pays) :

nom	iso	annee	cas	deces
Brésil	BR	2009	309316	85
Brésil	BR	2010	334667	76
Kenya	KE	2010	898531	26017
Mali	ML	2011	307035	2128
Ouganda	UG	2010	1581160	8431

- La table `demographie` recense la population totale de chaque pays ; certaines lignes de cette table sont données en exemple :

pays	periode	pop
BR	2009	193020000
BR	2010	194946000
KE	2010	40909000
ML	2011	14417000
UG	2010	33987000

1. Au vu des données présentées dans la table `palu`, parmi les attributs `nom`, `iso` et `annee`, quels attributs peuvent servir de clé primaire ? Un couple d'attributs pourrait-il servir de clé primaire ? (on considère qu'une clé primaire peut posséder plusieurs attributs). Si oui, en préciser un.

2. Écrire une requête SQL qui récupère depuis la table `palu` toutes les données de l'année 2010 qui correspondent à des pays où le nombre de décès dus au paludisme est supérieur ou égal à 1 000.

3. On appelle taux d'incidence d'une épidémie le rapport du nombre de nouveaux cas pendant une période donnée sur la taille de la population-cible pendant la même période. Il s'exprime généralement en « nombre de nouveaux cas pour 100 000 personnes par année ». Il s'agit d'un des critères les plus importants pour évaluer la fréquence et la vitesse d'apparition d'une épidémie.

Écrire une requête qui détermine le taux d'incidence du paludisme en 2011 pour les différents pays de la table `palu`.

4. Écrire une requête permettant d'afficher le nombre de nouveaux cas de paludisme en 2010 pour les différents pays de la table `palu`.

5. Écrire une requête permettant d'afficher le maximum de nouveaux cas de paludisme en 2010.

6. Écrire une requête permettant d'afficher le nom du pays ayant le plus grand nombre de nouveaux cas de paludisme en 2010 (on pourra supposer qu'il n'y a pas de pays ex aequo pour les nombres de cas).

7. Écrire une requête permettant de déterminer le nom du pays ayant eu le deuxième plus grand nombre de nouveaux cas de paludisme en 2010

8. On considère la requête R, qui s'écrit dans le langage de l'algèbre relationnelle :

$$R = \pi_{\text{nom,deces}}\left(\sigma_{\text{annee=2010}}\left(\text{palu}\right)\right)$$

On suppose que le résultat de cette requête a été converti en une table `deces2010` constituée de couples (chaîne, entier).

Écrire une requête SQL permettant de trier la liste `deces2010` par ordre croissant du nombre de décès dus au paludisme en 2010.

Analyse du problème

Cet exercice est extrait de l'épreuve d'informatique du concours Mines Ponts 2016. On utilise une rubrique imbriquée. Le résultat d'une requête imbriquée en SQL ne fournit qu'un champ dans cet exercice (maximum du nombre de nouveaux cas de paludisme en 2010).

Cours :

La sélection $\sigma_{\text{annee=2010}}\left(\text{palu}\right)$ s'applique à la table `palu` et permet d'extraire de celle-ci les éléments qui satisfont un critère de sélection $\left(\text{annee=2010}\right)$.

La projection $\pi_{\text{nom,deces}}\left(R'\right)$ s'applique à une relation R' et ne garde que les colonnes `nom`, `deces`. Contrairement à la sélection, la projection ne supprime pas des lignes mais des colonnes.

1. Une clé primaire sert à identifier une ligne de manière unique. Aucune colonne dans la table `palu` n'est une clé primaire.

Le couple (`iso`, `annee`) est une clé primaire.

2.

```
SELECT *
FROM palu
WHERE annee=2010
AND deces>=1000;
```

3.

```
SELECT nom, cas/pop*100000 AS 'taux incidence'
FROM palu
JOIN demographie ON iso=pays AND periode=annee
WHERE annee=2011;
```

4.

```
SELECT nom, cas
FROM palu
WHERE annee=2010;
```

On obtient : Brésil 334667 ; Kenya 898531 ; Ouganda 1581160.

5.

```
SELECT max(cas)
FROM palu
WHERE annee=2010;
```

6.

```
SELECT nom, cas
FROM palu
WHERE annee=2010 AND cas =(SELECT max(cas) FROM palu WHERE
annee=2010);
```

On utilise une rubrique imbriquée qui donne le maximum de nouveaux cas de paludisme en 2010.

7. Il faut créer deux rubriques imbriquées.

```
SELECT nom, cas
FROM palu
WHERE annee=2010 AND cas =
(SELECT max(cas) FROM palu WHERE annee=2010 AND
cas <(SELECT max(cas) FROM palu WHERE annee=2010));
```

8. On obtient alors une table `deces2010` contenant deux colonnes :

- `nom`, de type chaîne de caractères, désigne le nom du pays ;
- `deces`, de type entier, désigne le nombre de nouveaux cas de paludisme en 2010.

```
SELECT nom,deces
FROM deces2010
ORDER BY deces;
```

Partie 14

Algorithmique numérique

Plan

17

Algorithmique numérique (Spé) (uniquement TSI et TPC)

Exercice 17.1 : Équation différentielle du premier ordre

On considère l'équation différentielle : $\frac{dy}{dt} = \frac{E - y}{\tau}$ avec E et τ 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 yi est représenté par la liste `Y`.

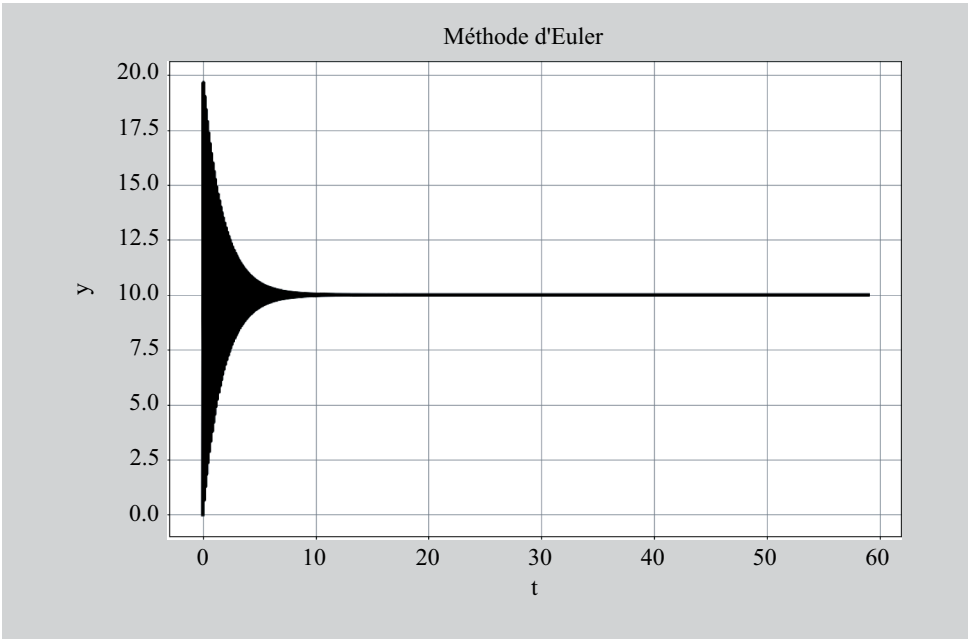
1. Écrire une relation de récurrence permettant de calculer y_{i+1} en fonction de y_i, E, h et τ.

2. 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`.

3. Écrire le programme principal permettant d'afficher graphiquement y en fonction du temps t avec les conditions suivantes : $t0 = 0$, $y(0) = 0$, $E = 10$, $h = 0{,}2$ ms, $\tau = 30$ ms et $N = 1\,000$. 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 ».

4. Commenter la courbe obtenue avec $h = 59$ ms, les autres conditions précédentes demeurant inchangées.



Analyse du problème

On utilise la méthode d'Euler pour résoudre numériquement une équation différentielle du premier ordre. On étudie l'influence du pas sur la qualité de la solution.

Cours :
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 $N-1\}$,
- $\{u_i = u\left(t = ih\right)$, 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 $N-1\}$,
- $\{U[i] = u_i = u\left(t = ih\right)$, 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 :

$$u\left(t_{i+1}\right) = u\left(t_i\right) + \left(t_{i+1} - t_i\right)\left(\frac{\mathrm{d}u}{\mathrm{d}t}\right)_{t_i} + \ldots$$

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.

1. 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.

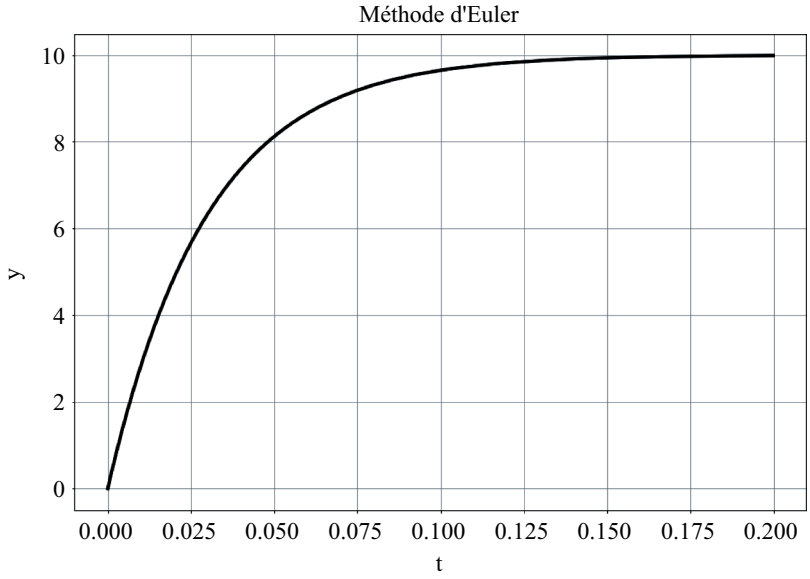
2.

```
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
```

3.

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



4. 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 : $\frac{\mathrm{d}y}{\mathrm{d}t} = \lambda y$. 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.

Exercice 17.2 : Équation différentielle du deuxième ordre

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 : $\frac{\mathrm{d}Y}{\mathrm{d}t} = F\left(Y(t), t\right)$. On pose $Y(0) = Y_0 = \begin{pmatrix} u(0) \\ g(0) \end{pmatrix}$.

La réponse $u(t)$ recherchée sur l'intervalle $\left[0, t_{\max}\right]$ 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 $T = \left[t_0 = 0, t_1, \ldots t_{N-1} = t_{\max}\right]$. Pour chaque instant t_i, une valeur approchée Y_i de la solution $Y\left(t_i\right)$ de l'équation différentielle est recherchée. On utilisera 3 listes `T`, `U` et `G` comportant chacune N éléments.

1. Déterminer la fonction $F\left(Y(t), t\right)$.

2. Donner la relation de récurrence qui lie Y_{i+1} à $Y_i, F\left(t_i, Y_i\right)$ et au pas de calcul h.

3. 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 $F\left(Y_i\left(t_i\right), t_i\right)$.

4. 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.

Analyse du problème

On cherche à résoudre numériquement une équation différentielle du deuxième ordre, on se ramène à deux équations différentielles du premier ordre. On utilise la méthode d'Euler explicite étudiée dans l'exercice précédent.

1. On pose $g(t)=\dfrac{\mathrm{d}u}{\mathrm{d}t}$. L'équation différentielle (1) s'écrit alors sous la forme : $\dfrac{\mathrm{d}g}{\mathrm{d}t}+bg+cu=0$, soit $\dfrac{\mathrm{d}g}{\mathrm{d}t}=-cu-bg$. On transforme alors l'équation différentielle (1) en un système de deux équations différentielles du premier ordre : $\begin{pmatrix}\dfrac{\mathrm{d}u}{\mathrm{d}t}\\ \dfrac{\mathrm{d}g}{\mathrm{d}t}\end{pmatrix}=\begin{pmatrix}g(t)\\ -cu(t)-bg(t)\end{pmatrix}$.

On a donc $\dfrac{\mathrm{d}Y}{\mathrm{d}t}=F\left(Y(t),t\right)$ 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 : $[g_i,-cu_i-bg_i]$.

2. Pour évaluer numériquement $\left(\dfrac{\mathrm{d}Y}{\mathrm{d}t}\right)_{t_i}$ à l'instant t_i, on utilise la méthode d'Euler explicite :

$$\left(\frac{\mathrm{d}Y}{\mathrm{d}t}\right)_{t_i}\approx\frac{Y(t_{i+1})-Y(t_i)}{h}$$

On a alors : $\dfrac{Y(t_{i+1})-Y(t_i)}{h}=F\left(Y(t_i),t_i\right)$. 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 $\dfrac{Y_{i+1}-Y_i}{h}=F\left(Y_{i+1},t_{i+1}\right)$. Dans ce cas, la nouvelle valeur Y_{i+1} est calculée en utilisant la valeur de la dérivée à la nouvelle position.

3. 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 : $[g_i,-cu_i-bg_i]$.

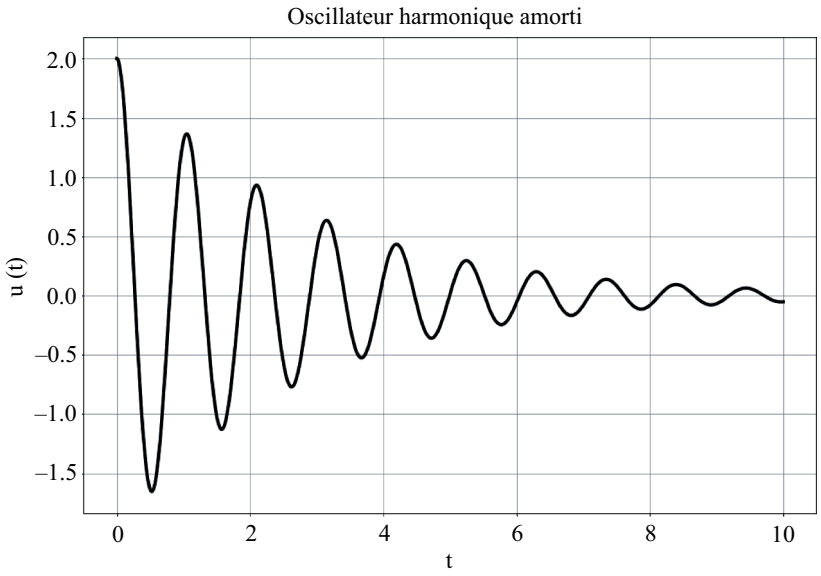
- On récupère u_i avec le premier élément de la liste `Yi` qui est en argument d'entrée de la fonction, soit $u_i = Y_i[0]$.
- On récupère g_i avec le deuxième élément de la liste `Yi` qui est en argument d'entrée de la fonction, soit $g_i = Y_i[1]$.

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

4. Il faut bien définir le nombre de points. Comme $t_{\max} = (N-1)h$, alors $N = \text{int}\left(1 + \dfrac{t_{\max}}{h}\right)$. 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 :



Exercice 17.3 : Résolution d'un système linéaire par la méthode de Gauss

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 injecté 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
	1 300	3,064	4,162	5,248	6,33	7,734	8,774	9,766
	1 700	3,27	4,412	5,552	6,644	7,734	8,774	9,766
	2 200	3,462	4,63	5,804	6,952	8,062	9,126	10,154
	2 700	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 M valeurs de pression à l'admission en bar, est stockée dans une liste de valeurs : $P = \left[P_j \right]_{0 \leq j \leq M-1}$.
- La première colonne (900 ; 1 300 ; 1 700 …), qui contient N valeurs de vitesse de rotation du moteur en tr/min, est stockée dans une liste de valeurs : $OMEGA = \left[OMEGA_i \right]_{0 \leq i \leq N-1}$.
- 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 $P[j] \leq Pcol \leq P[j+1]$ et $OMEGA[i] \leq OMEGAmot \leq OMEGA[i+1]$.

1. 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 : $A[id] \leq val \leq A[id+1]$. 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.

2. 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` :

$$ST = \Big[\big[T[i,j], T[i,j+1], [P[j]], OMEGA[i]\big],$$
$$\big[T[i+1,j], T[i+1,j+1], P[j+1], OMEGA[i+1]\big]\Big]$$

3. 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 :

$$t(x,y) = b_0 + b_1 x + b_2 y + b_3 xy \ ; \ x \in [0,1] \text{ et } y \in [0,1]$$

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 :

$$PQ = R \text{ et } Q = P^{-1}R$$

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.

4. Écrire l'algorithme du pivot de Gauss permettant d'obtenir la matrice Q. On admettra que les termes diagonaux sont tous non nuls.

5. 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 :

$$\begin{pmatrix} t(x=0,y=0) = ST[0,0] & t(x=1,y=0) = ST[0,1] \\ t(x=0,y=1) = ST[1,0] & t(x=1,y=1) = ST[1,1] \end{pmatrix}$$

6. Montrer que le problème à résoudre devient :

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix}=\begin{pmatrix} ST[0,0] \\ ST[0,1] \\ ST[1,0] \\ ST[1,1] \end{pmatrix}$$

7. Montrer que la résolution par substitution s'écrit :

$$\begin{aligned} b_0 &= ST[0,0] \\ b_1 &= ST[0,1]-ST[0,0] \\ b_2 &= ST[1,0]-ST[0,0] \\ b_3 &= ST[1,1]-ST[0,1]-ST[1,0]+ST[0,0] \end{aligned}$$

8. Comparer la complexité de cet algorithme à celui présenté à la question 5.

9. 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.

Analyse du problème

On utilise la méthode de Gauss avec recherche partielle du pivot pour résoudre un système linéaire d'ordre *n*.

Cours :

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 $i \neq p$:

- $i = 1$: $\text{valeur} = P'_{i=1,p=0} = 1$. 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 $R' = \begin{pmatrix} 0{,}5 \\ 0{,}5 \\ 1 \end{pmatrix}$.

- $i = 2$: $\text{valeur} = P'_{i=2,p=0} = 2$. D'où : $\text{ligne}(i=1) \leftarrow \text{ligne}(i=2) - \text{valeur} \times \text{ligne}(p=0)$. 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 : ligne$(i=0) \leftarrow$ ligne $(i=0) -$ valeur $\times$ 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 $P'_{i=2,p=2} = \dfrac{3}{5}$ 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 $= P'_{i=0,p=2} = \dfrac{3}{5}$. 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 $R' = \begin{pmatrix} \dfrac{1}{2} - \dfrac{3}{5} \times \dfrac{5}{6} \\ 0 \\ 5/6 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 5/6 \end{pmatrix}$.

- $i = 1$: valeur $= P'_{i=1,p=2} = \dfrac{1}{5}$. 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 :

$PQ = \begin{pmatrix} 2 & 4 & 2 \\ 1 & 4 & 2 \\ 2 & 9 & 3 \end{pmatrix} Q = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$

1.

```
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
```

2.

```
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
```

3.

```
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]]`.

4.

```
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)
```

5. 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 : $4+(n-1-p)\times 4 = 4n-4p$ 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 : $n\times(2+4n+4) = 6n+4n^2$ 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 $O\left(n^3\right)$.

6. On pose : $t(x,y) = b_0+b_1x+b_2y+b_3xy$; $x\in[0,1]$ et $y\in[0,1]$.

On en déduit que : $t(0,0)=b_0$; $t(1,0)=b_0+b_1$; $t(0,1)=b_0+b_2y$ et $t(x,y)=b_0+b_1+b_2+b_3$.

On a bien $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_0+b_1 \\ b_0+b_2 \\ b_0+b_1+b_2+b_3 \end{pmatrix} = \begin{pmatrix} t(0,0) \\ t(1,0) \\ t(0,1) \\ t(1,1) \end{pmatrix} = \begin{pmatrix} ST[0,0] \\ ST[0,1] \\ ST[1,0] \\ ST[1,1] \end{pmatrix}$.

7. On résout directement ce système :

$b_0 = ST[0,0]$; $b_1 = ST[0,1] - b_0 = ST[0,1] - ST[0,0]$; $b_2 = ST[1,0] - b_0 = ST[1,0] - ST[0,0]$.

$b_3 = ST[1,1] - b_0 - b_1 - b_2 = ST[1,1] - ST[0,0] - (ST[0,1] - ST[0,0]) - (ST[1,0] - ST[0,0])$

Après simplification, on a : $b_3 = ST[1,1] + ST[0,0] - ST[0,1] - ST[1,0]$.

On obtient finalement :

$$b_0 = ST[0,0]$$
$$b_1 = ST[0,1] - ST[0,0]$$
$$b_2 = ST[1,0] - ST[0,0]$$
$$b_3 = ST[1,1] - ST[0,1] - ST[1,0] + ST[0,0]$$

8. 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 5 puisqu'on a une matrice triangulaire.

9.

```
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)
```

Exercice 17.4 : Interpolation polynomiale de Lagrange

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 et le polynôme d'interpolation de degré n par rapport aux n+1 points équidistants dans l'intervalle $[-5,5]$. 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`.

1. Écrire une fonction `lagrange` qui admet comme arguments un entier `i`, un réel `x` et un entier `n`. La fonction retourne $L_i(x)=\prod_{\substack{j=0\\ j\neq i}}^{n}\dfrac{x-x_j}{x_i-x_j}, i\in[\![0,n]\!]$.

2. Écrire une fonction `poly` qui admet comme arguments un réel `x` et un entier `n`. La fonction retourne $p(x)=\sum_{i=0}^{n} f_i L_i(x),\ x\in\mathbb{R}$.

3. Écrire le programme principal permettant de représenter graphiquement la fonction f et le polynôme d'interpolation de Lagrange dans l'intervalle [–5, 5] pour n = 4. Que se passe-t-il pour n = 10 et n = 14 ?

Analyse du problème

On détermine les n+1 composantes du polynôme d'interpolation qui passe par les n+1 points imposés. Le phénomène de Runge apparaît lorsque n est trop grand.

1.

```
def lagrange(i, x, n):
    # i entier
    # x réel, n entier
    Li=1
```

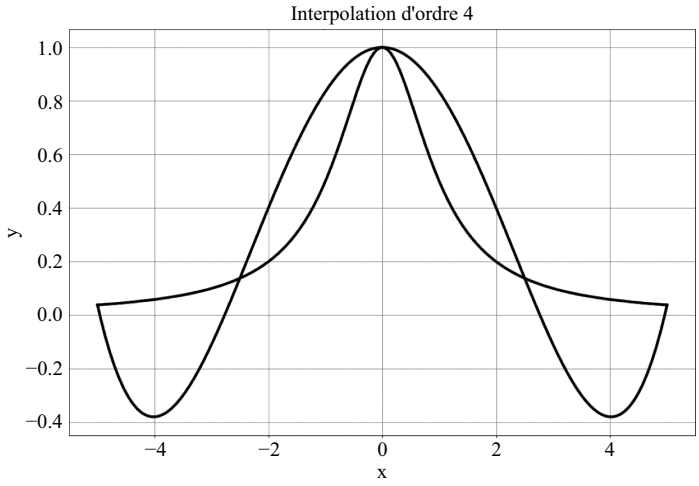
```
    for j in range(n+1): # j varie entre 0 inclus et n+1 exclu
        if j!=i:
            Li*=(x-xs[j])/(xs[i]-xs[j])
    return Li
```

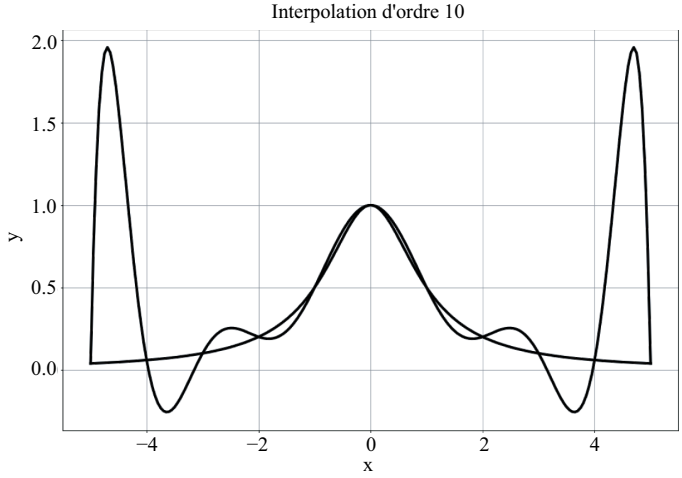
2.

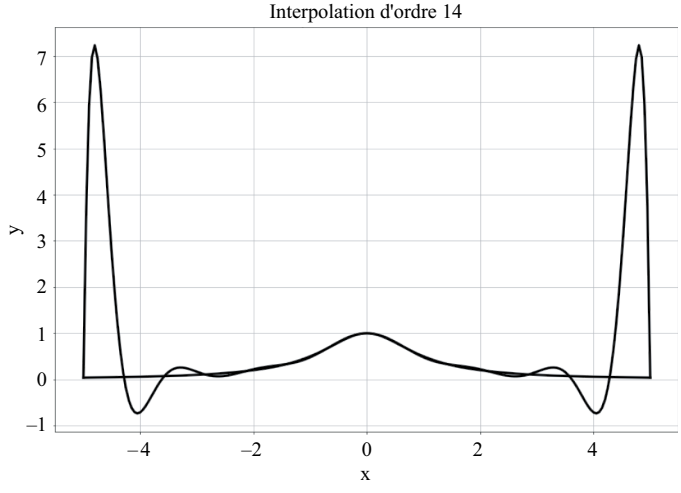
```
def poly(x, n):
    # x réel, n entier
    som=0
    for i in range(n+1): # i varie entre 0 inclus et n+1 exclu
        som+=ys[i]*lagrange(i, x, n)
    return som
```

3.

```
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=4
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
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
```


Interpolation d'ordre 4
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
y
−4
−2
0
2
4
x


Interpolation d'ordre 10
2.0
1.5
1.0
0.5
0.0
y
−4
−2
0
2
4
x


Interpolation d'ordre 14
7
6
5
4
3
2
1
0
−1
y
−4
−2
0
2
4
x

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 suivant « Interpolation par morceaux »).

Exercice 17.5 : Interpolation par morceaux

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 :

$$p_1^h(x)=p_{1,i}(x), \forall x\in I_i$$

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`.

1. Donner l'expression de $p_{1,i}$ sur chaque intervalle I_i, pour $i\in[\![0,n-1]\!]$.

2. 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 $[a,b]$.

Analyse du problème

Pour résoudre le problème d'oscillations qui apparaît lorsque n est trop grand, on utilise une interpolation par morceaux.

1. 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 $p_{1,i}(x)=y_i+(x-x_i)\dfrac{y_{i+1}-y_i}{x_{i+1}-x_i}$.

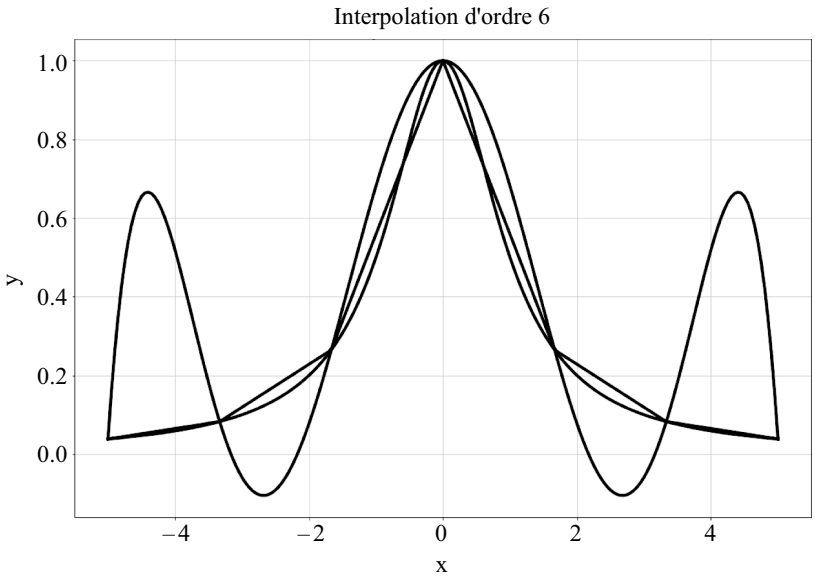
2. On pose $h = \dfrac{b-a}{n}$ et $x_i = a + ih, i \in [\![0, n]\!]$.

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