Les listes

Cours sur les listes

Rappel sur les boucles :

1) Boucle `while`

On continue une série d'instructions tant qu'une condition est vérifiée.

Exemple :

```
""" donne le nombre de caractères
de la première phrase d'un texte."""
Phrase  = "Bonjour monsieur Blanc. Quand faisons-nous les projets?"
compteur = 0
car = Phrase[compteur]
while not(car == "."):
    compteur += 1
    car = Phrase[compteur]
print(compteur)
```

2) Boucle `for`

Typiquement : on itère sur une liste et pour chaque élément de cette liste, on effectue une instruction.

Exemple :

```
# compter le nombre de nombres pairs d'une liste L d'entiers
def nbPairs(L):
    compt = 0
    for nb in L:
        if not(nb%2):
            compt += 1
    return compt
```

On peut aussi travailler sur tout objet itérable, comme les chaînes de caractères ou les tuples.

Exemple :

```
# A est une chaîne, B est un tuple.
A = "bonjour"; B = 2,8,9,16 # ou B = (2,8,9,16)
for a in A:
    print(a)
for b in B:
    print(b)
```

On utilise souvent une boucle for pour simplement répéter une instruction ou pour itérer sur les indices d'un objet itérable. Dans ce cas, on utilise `for k in range(n)` où n est la taille de l'objet. On rappelle que `range(n)`représente un objet itérable formé des entiers de 0 à $n-1$.
Exemple :

```
1 Chaine = "Je suis en BCPST2"
2 for k in range(len(Chaine)):
3     print("le "+str(k+1)+" ième caractère est "+Chaine[k])
4
5 # version équivalente
6 for k in range(len(Chaine)):
7     print("le {} ième caractère est {}".format(str(k+1),Chaine[k]))
```

Les Listes :

1) Créer une liste

- Création par énumération :

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

- Création par concaténation :

```
A = [2,3]
B = ["bonjour","jean",198]

#On met bout à bout les listes A et B et on stocke le résultat dans M:
M = A+B
L = [0]*10  # Liste de 10 nombres 0.
L = [0,1]*50 # liste où la séquence 0,1 est répétée 50 fois.
```

- Utilisation de "range" :

```
L1 = list(range(50))  # liste des 50 entiers de 0 à 49
L2 = list(range(3,12)) # liste des entiers de 3 à 11
L3 = list(range(5,15,3))
""" Liste des entiers de 5 à 15 (exclu) avec un pas de 3.
Ici L3 = [5,8,11,14]."""
```

- Création par compréhension :

```
L = [i**2 for i in range(100)]
```

(crée la liste [0,1,4,9,...] des carrés des entiers 0, 1, 2,..., 99)

2) Opérations sur les listes

On considère une liste L :

Appel des éléments de L

Commande	Effet
len(L)	donne la taille de L
L[i]	élément d'indice i de L (les indices commencent à 0)
L[-1]	dernier élément de la liste L
L[-2]	avant dernier élément de la liste L
L[i :j]	sous-liste contenant les éléments d'indice i (inclus) à j (exclus)
L[i :]	sous-liste contenant les éléments d'indice i jusqu'à la fin
L[:i]	sous-liste contenant les éléments du début jusqu'à celui d'indice i (exclu)

Opérations sur une liste L

Commande	Effet
L.append(v)	ajoute un élément v à la fin de la liste L
L.extend(s)	ajoute les éléments de s à la fin de la liste L
L.insert(i,v)	insère l'objet v à l'indice i dans L
L.pop()	supprime le dernier élément de L et retourne l'élément supprimé
L.pop(i)	supprime et renvoie l'élément d'indice i de L
L.remove(v)	supprime la première valeur v dans L
L.reverse()	inverse l'ordre des éléments de L
L.sort()	trie la liste L
del L[i], del L[i :j]	supprime un ou des éléments de L

Tableaux empruntés ici : http://bcpst.parc.free.fr/joomla/DOCUMENTS/Maths952/Info/AideMemoire.pdf

Voyons maintenant la procédure générale pour construire une liste.

Point méthode

Pour construire une liste, on dispose essentiellement de trois méthodes.

❶ Si on connaît la taille de la liste à construire, on peut l'initialiser comme une liste de la bonne taille remplie de zéros puis on modifie ses coefficients.

Ci-dessous, on crée par exemple une liste de 100 entiers compris entre 10 et 20 :

```
Liste1 = [0]*100
for k in range(100):
        Liste1[k] = randint(10,20)
```

❷ Si on ne connaît pas à l'avance la taille de la liste à construire, on l'initialise comme une liste vide, puis on place en fin de liste les éléments qui la constitue à l'aide de la méthode append().

Exemple : la création de la liste des entiers compris entre 0 et 1000 (inclus) divisibles par 3 mais pas par 5.

```
Liste2 = []
for k in range(1001):
        if k%3 == 0 and k%5 != 0:
                Liste2.append(k)
```

```
#Bis
Liste2 = []
for k in range(1001):
        if not(k%3) and k%5:
                Liste2.append(k)
```

❸ Dans les cas relativement simples, on peut directement construire la liste par compréhension. On reprend les deux exemples précédents pour illustrer cela.

```
Liste1 = [randint(10,20) for k in range(100)]
Liste2 = [k for k in range(1001) if k%3==0 and k%5!=0]
```

3) Recopier une liste

Remarque

Si vous avez besoin de faire une copie d'une liste A dans une liste B et de modifier celle-ci de façon indépendante de la liste d'origine, une instruction du style "B=A" ne correspondra pas à ce qui est attendu : en effet, toute modification ultérieure sur A sera également effectuée sur B et vice-versa.

Il faut donc procéder ainsi :
Copie de liste

```
B = list(L)
```

4) Tableau à double entrée ou matrice

Il n'y a pas de structure en Python pour les matrices ou tableau à plusieurs entrées. On peut cependant obtenir des objets similaires en créant des listes de listes.
Pour obtenir un affichage sous forme de matrice, il faut utiliser le package numpy. Ce package permet d'ailleurs de faire beaucoup de calculs sur les matrices.
Création d'une matrice (en supposant les variables i et j affectées) :

```
import numpy as np
# liste de 4 sous-listes formées de 6 zéros chacune:
M = [[0]*6 for i in range(4)]
print(M[i][j]) # affichage du j-ème élément de la i-ème sous-liste.
# conversion de M en matrice à 4 lignes et 6 colonnes:
Mat = np.array(M) # on stocke la matrice dans la variable Mat.
# affichage du coefficient à la i-ème ligne, j-ème colonne:
print(M[i,j])
```

Remarque

Ne pas oublier qu'en Python, la numérotation commence à 0.

Remarque : copie d'une matrice

On rencontre la même difficulté pour copier une matrice ou une liste de listes. Pour copier de tels objets, on procèdera ainsi :

Copie de matrice :

```
from copy import deepcopy
B = deepcopy(M)
MatBis = deepcopy(Mat)
```

5) Exemples à connaître sur les listes

Recherche d'un élément (cible) dans une liste :

```
def Recherche(Liste, cible):
        for elem in Liste:
                if elem == cible:
                        return True
        return False
```

```
def RechercheAvecIndice(Liste, cible):
        for k in range(len(Liste)):
                if Liste[k] == cible:
                        return k
        return False
```

Recherche du maximum d'une liste de nombre (également avec l'indice d'un élément réalisant le maximum) :

```
def RechercheMax(Liste):
        Max = -float("inf")
        for elem in Liste:
                if elem > Max:
                        Max = elem
        return Max
```

```
def RechercheMaxAvecIndice(Liste):
        Max = -float("inf")
        for k in range(len(Liste)):
                if Liste[k] > Max:
                        Max = Liste[k]
                        indice = k
        return Max,indice
```

Calcul de la moyenne d'une liste de nombres

```
def CalculMoyenne(Liste):
        Somme = 0
        for elem in Liste:
                Somme += elem
        return Somme/len(Liste)
```

Calcul de la variance d'une liste de nombres $[x_1, x_2, \ldots, x_n]$. D'après la formule de KÃ¶nig-Huygens, c'est le nombre $\frac{1}{n}\left(\sum_{i=1}^{n} x_i^2\right) - \left(\frac{1}{n}\sum_{i=1}^{n} x_i\right)^2$ (différence de la moyenne des carrés et du carré de la moyenne).

```
def CalculVariance(Liste):
        taille = len(Liste)
        Somme1, Somme2 = 0, 0
        for elem in Liste:
                Somme1 += elem
                Somme2 += elem**2
        Moy1, Moy2 = Somme1/taille, Somme2/taille
        return Moy2 - Moy1**2
```

Extraction des chiffres d'un nombre dans le système décimal (on stocke les chiffres dans une liste) :

```
def Chiffres(nombre):
        ListeChiffres = []
        while nombre > 0:
                chiffre = nombre%10
                ListeChiffres = [chiffre] + ListeChiffres
                nombre -= chiffre
                nombre //= 10
        return ListeChiffres

# Exemple: avec 1645
""" La première itération donnera
chiffre = 5
ListeChiffres = [5]
nombre = 1640
nombre = 164
puis la seconde itération donnera
chiffre = 4
ListeChiffres = [4,5]
nombre = 160
nombre = 16
etc.
"""
```

Procédure inverse :

```
def Nombre(Chiffres):
        nb = 0
        for k in range(len(Chiffres)):
                nb += Chiffres[-k-1]*10**k
        return nb
```

6) Les piles

Une pile est une structure de données permettant de stocker un certain nombre d'éléments, puis de les récupérer plus tard. Les éléments sont extraits de la pile selon la règle "dernier entré, premier sorti", plus connue en anglais sous le nom "last in, first out", soit LIFO : un élément stocké dans une pile en est toujours extrait avant les éléments qui y étaient depuis plus longtemps.

Avec une pile, on effectue essentiellement deux opérations :
empiler une valeur au sommet de la pile ou dépiler la valeur située au sommet.

Implantation :
- Pour initialiser une pile, on peut créer une liste vide :

```
Pile = []
```

- Pour empiler une valeur, il suffit d'utiliser la méthode append() :

```
Pile.append(valeur)
```

- Pour dépiler la valeur située au sommet, il suffit d'utiliser la méthode pop() (qui permet aussi de récupérer cette valeur) :

```
valeur = Pile.pop()
```

- Pour afficher la valeur se trouvant au sommet :

```
print(Pile[-1])
```

- Pour afficher le nombre d'éléments de la pile :

```
print(len(Pile))
```

Exercices sur les listes

Exercice 1 :

1) Écrire une fonction Python prenant en entrée une liste L de nombres entiers et un entier a. La fonction doit donner en sortie le nombre de fois où a figure dans la liste L.

2) Écrire une fonction Python prenant en entrée une liste L de nombre entiers compris entre 0 et 100 (inclus). La fonction doit donner en sortie une liste M indiquant, pour chaque entier a compris entre 0 et 100, le nombre de fois où ce nombre apparaît dans L (M doit être une liste *de comptage* : l'élément d'indice a de M doit être égal au nombre de fois où a apparaît dans L).

Exercice 2 :

1) Écrire une fonction Python prenant en entrée une liste $\mathsf{L} = [a_0, a_1, a_2, \ldots, a_{n-1}]$ de n éléments et donnant en sortie la liste LL à $2n$ éléments définie par $\mathsf{LL} = [a_0, 0, a_1, 0, a_2, 0, \ldots, a_{n-1}, 0]$ (autrement dit LL est obtenue en intercalant des zéros entre les éléments de L, y compris à la fin).

2) Écrire une fonction Python prenant en entrée deux listes $\mathsf{L} = [a_0, a_1, a_2, \ldots, a_{n-1}]$ et $\mathsf{M} = [b_0, b_1, b_2, \ldots, b_{n-1}]$ à n éléments. La fonction doit renvoyer en sortie la liste $\mathsf{LM} = [a_0, b_0, a_1, b_1, \ldots, a_{n-1}, b_{n-1}]$ obtenue en "mélangeant" les deux listes L et M. La fonction devra également retourner False dans le cas où les deux listes L et M n'auraient pas le même nombre d'éléments.

Exercices sur les piles :

Exercice 3 : (*Jeu de carte*)

Deux joueurs disposent chacun de cartes numérotées de 1 à 9. Un tour du jeu se déroule ainsi : chaque joueur découvre la carte au sommet de son tas. La carte de plus grande valeur remporte le tour : le joueur qui gagne récupère sa carte et son tas initial et place au-dessus la carte de son adversaire. Si les deux cartes sont de même hauteur, les deux joueurs retirent ces cartes de leur jeu. Le jeu continue ainsi jusqu'à épuiser l'un des deux tas de cartes (si possible).

Écrire une fonction Python prenant en entrée deux listes $L1$ (le tas du joueur 1) et $L2$ (le tas du joueur 2), un entier $p \geqslant 0$ (le nombre de tours), et donnant en sortie les listes représentant les tas des deux joueurs après p tours si le jeu n'est pas terminé. Si le jeu se termine avant p tours, on affichera l'état des deux tas à la fin de la partie.

Exercice 4 :

1) Écrire une fonction Python prenant en entrée deux listes d'entiers $L1$ et $L2$ et donnant en sortie la liste formée des éléments communs à $L1$ et $L2$.

2) En utilisant la notion de pile, écrire une seconde version de cette fonction (on triera au préalable les deux listes grâce à la méthode `sort()`) . Quelle méthode nécessite le plus d'opérations si les deux listes sont déjà triées ?

Corrigés des exercices sur les listes

Exercice 1 (Corrigé)

1)

```python
# Exercice 1, question 1:
def NbOccurrences(L,a):
        Nb = 0
        for elem in L:
                if elem == a:
                        Nb += 1
        return Nb
```

2)

```python
# Exercice 1, question 2:
def ListeOccurrences(L):
        M = [0]*101
        for a in L:
                M[a] += 1
        return M
```

Exercice 2 (Corrigé)

1)

```python
# Exercice 1, question 1:
def Intercale(L):
        NouvL = []
        for k in range(len(L)):
                NouvL.append(L[k])
                NouvL.append(0)
        return NouvL
```

2)

```python
# Exercice 1, question 2:
def Melange(L,M):
        NouvL = []
        if len(L) != len(M):
                return False
        for k in range(len(L)):
                NouvL.append(L[k])
                NouvL.append(M[k])
        return NouvL
```

Exercice 3 **(Corrigé)**

On s'arrête prématurément si un des deux tas est vide.

```
def Jeu(L1,L2,p):
    tour = 0
    while tour < p and len(L1)*len(L2) != 0:
        carte1 = L1[-1]
        carte2 = L2[-1]
        if carte1 > carte2:
            L1.append(L2.pop())
        elif carte2 > carte1:
            L2.append(L1.pop())
        else: # cas d'égalité
            L1.pop(); L2.pop()
        tour += 1
    return L1,L2
```

Exercice 4 **(Corrigé)**

1) On fait deux boucles for imbriquées pour comparer deux à deux les éléments des deux listes. Parallèlement, on crée une liste `Communs` des éléments en commun. On ne rajoute pas une valeur commune déjà répertoriée dans `Communs`.

```
def ElemCommuns(L1,L2):
    Communs = []
    for elem1 in L1:
        for elem2 in L2:
            if elem1 == elem2:
                EstDejaDansCommuns = False
                for elem in Communs:
                    if elem == elem1:
                        EstDejaDansCommuns = True
                if not(EstDejaDansCommuns):
                    Communs.append(elem1)
    return Communs
```

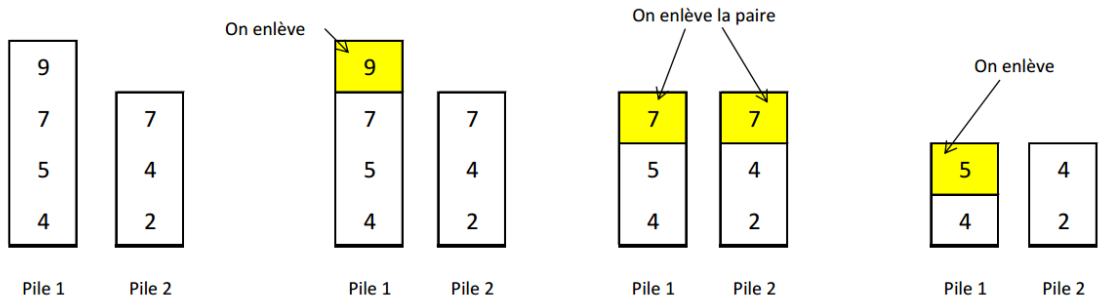
2)

L'idée est de trier les deux tableaux, et de les parcourir en parallèle. Si on place les valeurs dans deux piles, il y a trois cas possibles :

- la valeur en haut de la première pile est la plus grande, on l'élimine ;
- la valeur en haut de la seconde pile est la plus grande, on l'élimine ;

- les valeurs en haut de chaque pile sont identiques, on les enlève ensemble et on prend en compte cela pour une valeur dans l'intersection.

L'exemple suivant illustre les opérations précédentes :



Dès qu'une des deux piles est vide, on peut s'arrêter car il n'y aura plus aucune autre valeur dans l'intersection.
Pour l'implémentation : tant que les deux piles sont non vides, on compare les valeurs aux sommets des piles et on fait ce qu'il faut en fonction de la situation.

```
def ElemCommunsBis(L1,L2):
    L1.sort(); L2.sort()
    Communs = []
    # traitement des piles
    while len(L1) > 0 and len(L2) > 0:
        sommet1, sommet2 = L1[-1], L2[-1]
        if sommet1 > sommet2:
            L1.pop()
        elif sommet1 < sommet2:
            L2.pop()
        else: # cas où les deux éléments aux sommets des piles sont identiques
            Communs.append(sommet1)
            L1.pop(); L2.pop()
    return Communs
```

On note Na et Nb les tailles des listes L1 et L2 (que l'on suppose déjà triées). La fonction ElemCommunsBis utilise au plus Na + Nb opérations.
La fonction ElemCommuns utilise deux `for` imbriquées qui bouclent sur chacune des deux listes. Il y a donc au moins NaNb opérations (sans compter la troisième boucle `for` imbriquée qui parcourt `Communs`) et donc cette dernière fonction est moins efficace que ElemCommunsBis (car nécessite plus d'opérations).

Les matrices

Cours sur les matrices

Calcul matriciel

Ci-dessous, on crée les matrices $A = \begin{pmatrix} 1 & 3 & -1 \\ 2 & 0 & 5 \\ 4 & 2 & 2 \end{pmatrix}$, $B = \begin{pmatrix} 1 & 3 & -1 \\ 0 & 4 & 5 \end{pmatrix}$, $I_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ et on calcule successivement :
$3.A$, la somme des éléments de la première ligne de A, le rang de A, la matrice inverse de A, la taille de B, la transposée de B, la matrice $5.A + 8.{}^tA$, la matrice $B \times A$ et enfin la matrice A^5.
On utilise les deux bibliothèques numpy et sympy.

```
import numpy as np

A = np.array([[1,3,-1],[2,0,5],[4,2,2]])
B = np.array([[1,3,-1],[0,4,5]])
I = np.identity(3,int)

3*A
A[0,0]+A[0,1]+A[0,2]
np.linalg.matrix_rank(A) # rang de A
np.linalg.inv(A) # matrice inverse de A
B.shape
B.T # transposée de B
5*A+8*A.T
np.dot(B,A) # produit de B et A
np.linalg.matrix_power(A,5) # Calcul de A^5.
```

```
from sympy import *
init_printing(use_unicode=True)

A=Matrix(3,3,[1,3,-1,2,0,5,4,2,2])
B=Matrix(2,3,[1,3,-1,0,4,5])
I = eye(3)

3*A
A[0,0]+A[0,1]+A[0,2]
A.rank()
A.inv()
B.shape
B.T
5*A+8*A.T
B*A
A**5
```

Avec la bibliothèque sympy, on peut créer des matrices dépendant d'un ou plusieurs paramètres. Ci-dessous, on calcule C^2 où C est donnée par $C = \begin{pmatrix} 2 & a & -1 \\ 2 & 1 & 3 \\ b & 0 & -7 \end{pmatrix}$ avec a, b des paramètres inconnus.

```
a, b = symbols('a b')
C = Matrix(3,3,[2,a,-1,2,1,3,b,0,-7])
C**2
```

Remarque

On rencontre la même difficulté pour copier une matrice ou une liste de listes. Pour copier de tels objets, on procèdera ainsi :

Copie de matrice :

```python
# Avec numpy
import numpy as np
from copy import deepcopy

MatBis = deepcopy(Mat)
```

```python
# Avec Sympy

from sympy import *
MatBis = Mat[:,:]
```

Construction d'une matrice de taille quelconque

L'idée est d'initialiser la matrice à zéro (tous les coefficients sont nuls) puis de voir quelle formule détermine le coefficient en fonction des indices de la ligne et de la colonne où il se trouve.

Exemple 1 :

Pour créer les matrices carrées ci-dessous (de taille n),

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}, \quad B = \begin{pmatrix} 0 & 1 & 1 & \cdots & 1 \\ 0 & 0 & 1 & \ddots & 1 \\ 0 & 0 & 0 & \ddots & 1 \\ 0 & 0 & 0 & \ddots & 1 \\ 0 & 0 & 0 & \cdots & 0 \end{pmatrix}$$

(B est strictement triangulaire supérieure)

on remarque que $A[i,j] = 1$ si et seulement si $|i-j| = 1$ (et $A[i,j] = 0$ sinon), que $B[i,j] = 1$ si et seulement si $j > i$, et on peut donc utiliser les codes suivants :

```python
import numpy as np

def matrice_A(n):
    A = np.zeros((n,n))
    for i in range(n):
        for j in range(n):
            if abs(i-j) == 1:
                A[i,j] =  1
    return A
```

```python
import numpy as np

def matrice_B(n):
    B = np.zeros((n,n))
    for i in range(n):
        for j in range(n):
            if j > i:
                B[i,j] = 1
    return B
```

```python
import numpy as np

# plus rapide
def matrice_B_bis(n):
    B = np.zeros((n,n))
    for i in range(n):
        for j in range(i+1,n):
                B[i,j] = 1
    return B
```

Exemple 2 :

Création de la matrice $\begin{pmatrix} 1 & 2 & \cdots & n-2 & n-1 & n \\ 2 & 3 & \cdots & n-1 & n & 1 \\ 3 & 4 & \cdots & n & 1 & 2 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ n & 1 & \cdots & n-3 & n-2 & n-1 \end{pmatrix}$. Ici, le coefficient à la ligne `i` et colonne `j` est `(i+j)%n+1` (`(i+j)%n` est le reste de la division euclidienne de $i+j$ par n. Rappel : $0 \leqslant i \leqslant n-1$ et $0 \leqslant j \leqslant n-1$).

```python
import numpy as np

def MatGen(n):
    A = np.zeros((n,n))
    for i in range(n):
        for j in range(n):
            A[i,j] = (i+j)%n+1
    return A
```

```python
from sympy import *

def MatGenBis(n):
    return Matrix(n,n,lambda i,j:(i+j)%n+1)
```

Concaténation de deux matrices

`append(M,N,axis=0)` permet d'ajouter une matrice N après la dernière ligne de M.

`append(M,N,axis=1)` permet d'ajouter une matrice N après la dernière colonne de M.

Exercices sur les matrices

Exercice 1 :

1) Ecrire une fonction Python qui prend en entrée une matrice carrée M et qui renvoie en sortie la matrice ${}^tM$ (matrice transposée de M).

On impose d'écrire cette fonction "à la main".

2) On rappelle (ou on signale) que toute matrice carrée M se décompose de manière unique sous la forme $M = S + A$ où S est une matrice symétrique et A une matrice antisymétrique. On appelle S *la composante symétrique* de M et A *la composante antisymétrique* de M.

a) Rappeler les expressions de S et A en fonction de M et ${}^tM$.

b) En déduire une fonction Python qui prend en entrée une matrice carrée M et affiche en sortie les composantes symétriques et antisymétriques de M.

Exercice 2 :

Ecrire une fonction Python qui, à partir de $n \in \mathbb{N}$ tel que $n \geqslant 2$, renvoie en sortie le tableau carré T de taille $(n,\ 2n-1)$ formé de chiffres 1 formant un triangle à n lignes et de 0 partout ailleurs suivant les exemples ci-dessous :

Pour $n = 2$: $\begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$. Pour $n = 3$: $\begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$.

Pour $n = 4$:, $\begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$, etc.

Exercice 3 : Triangle de Pascal

1) Écrire une fonction Python prenant un entier naturel n et donnant en sortie la matrice à n lignes et n colonnes ci-dessous :

$$\begin{pmatrix} 1 & 0 & \cdots & 0 \\ 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & 0 & \cdots & 0 \end{pmatrix}$$

(que des zéros sauf sur la première colonne où se trouvent des 1).

2) Écrire une fonction Python permettant d'afficher, à partir de $m \in \mathbb{N}^*$, les m premières lignes du triangle de Pascal (sous forme de matrice). Par exemple,

pour $m = 2$: $\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$, pour $m = 3$: $\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 2 & 1 \end{pmatrix}$,

pour $m = 4$: $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 2 & 1 & 0 \\ 1 & 3 & 3 & 1 \end{pmatrix}$, pour $m = 5$: $\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 2 & 1 & 0 & 0 \\ 1 & 3 & 3 & 1 & 0 \\ 1 & 4 & 6 & 4 & 1 \end{pmatrix}$

Le coefficient à la i-ième ligne et j-ième colonne est $\binom{i}{j}$ si on numérote lignes et colonnes à partir de 0. D'après la formule de Pascal $\binom{i}{j} = \binom{i-1}{j} + \binom{i-1}{j-1}$, chaque coefficient du tableau (en dehors de la première colonne) est la somme de celui situé sur la ligne et colonne précédentes et de celui situé sur la même colonne et la ligne précédente.

Exercice 4 : Relations d'amitié (exercice emprunté au site http://www.france-ioi.org/)
(dans cet exercice, on prendra bien garde à la numérotation des indices en Python, à partir de 0.)

Sur un certain réseau social (dont nous tairons le nom), il est possible de se faire beaucoup d'amis. On considère un échantillon de n inscrits sur le site ($n \in \mathbb{N}^*$), repérés chacun par leur identifiant (un nombre entier entre 0 et $n-1$). Pour cet échantillon de personnes, on notera M la matrice (carrée de taille n) de relations d'amitié définie par :

$\forall (i,j) \in [\![0, n-1]\!]^2$, $\begin{cases} M[i,j] = 1 \text{ si l'inscrit } n^0 i \text{ est ami avec l'inscrit } n^0 j. \\ M[i,j] = 0 \text{ si l'inscrit } n^0 i \text{ et l'inscrit } n^0 j \text{ ne sont pas (encore) amis} \end{cases}$

On considérera que si l'inscrit i est ami avec l'inscrit j, alors l'inscrit j est ami avec l'inscrit i (ainsi, la matrice M est symétrique) et qu'une personne n'est jamais amie avec elle-même.

Par exemple, si $n = 3$ et $M = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$, cela signifie que les inscrits 0 et 2 sont amis, de même pour les inscrits 1 et 2 mais les inscrits 0 et 1 ne sont pas amis.

On ne considérera donc que les relations d'amitié au sein de l'échantillon de n personnes.
Toutes les fonctions Python qui seront écrites par la suite prendront (**entre autres**) en paramètre d'entrée la matrice d'amitié M. Si besoin est, il est possible dans chaque question de faire appel aux fonctions définies dans les questions précédentes.

1) Ecrire une fonction Python qui, à partir de M, n et $i \in [\![0, n-1]\!]$ détermine le nombre d'amis de l'inscrit i.
2) Ecrire une fonction Python qui, à partir de M, détermine une personne qui a le plus d'amis (renvoyer en sortie le numéro $j \in [\![0, n-1]\!]$ de son identifiant).
3) Ecrire une fonction Python qui, à partir de M, n et (i, j) dans $[\![0, n-1]\!]^2$, détermine le nombre d'amis qu'ont en commun les inscrits i et j.
4) Ecrire une fonction Python qui, à partir de M, n et $i \in [\![0, n-1]\!]$, détermine le nombre d'amis des amis de l'inscrit $n^0 i$ qui ne sont pas encore amis avec celui-ci.

Exercice 5 : Jeu de fléchettes

On considère des cibles comportant n zones se présentant sous la forme présentée ci-dessous au travers d'exemples :

Pour n = 3, $\begin{array}{|ccccc|}\hline 0&0&0&0&0\\0&1&1&1&0\\0&1&2&1&0\\0&1&1&1&0\\0&0&0&0&0\\\hline\end{array}$. Pour n = 4, $\begin{array}{|ccccccc|}\hline 0&0&0&0&0&0&0\\0&1&1&1&1&1&0\\0&1&2&2&2&1&0\\0&1&2&3&2&1&0\\0&1&2&2&2&1&0\\0&1&1&1&1&1&0\\0&0&0&0&0&0&0\\\hline\end{array}$

1) Ecrire une fonction Python permettant d'afficher (sous forme de tableau) une telle cible en fonction de n.

2) Un joueur envoie une flèche au hasard sur la cible. Le numéro de la zone où tombe sa flèche détermine le nombre de ses points.

Ecrire une fonction Python qui prend en entrée deux entiers naturels n et p et renvoie en sortie le nombre total des points obtenus par le joueur au cours de p lancers sur une cible à n zones.

Corrigés des exercices sur les matrices

Exercice 1 :

1)

```python
# Question 1
import numpy as np

def transpose(M):
    n = M.shape[0]
    N = np.zeros((n,n))
    for lig in range(n):
        for col in range(n):
            N[lig,col]=M[col,lig]
    return N
```

2) a) $S = \frac{1}{2}\left({}^tM + M\right)$ et $A = \frac{1}{2}\left(M - {}^tM\right)$.

b)

```python
# Question 2 b
def composantes(M):
    S = (M + transpose(M))/2
    A = (M - transpose(M))/2
    return S, A
```

Exercice 2 :

En numérotant les colonnes à partir de 0 (ce qui correspond à la convention d'indexation de Python), on s'aperçoit que le 1 médian se situe à la $(n-1)$-ième colonne. Si, pour un coefficient, l'écart entre l'indice j de sa colonne et l'indice $(n-1)$ de la colonne médiane est inférieur ou égal à son indice i de ligne, alors ce coefficient doit être égal à 1 et il est nul sinon. Cela nous inspire donc la fonction ci-dessous :

```python
import numpy as np

def UneMatrice(n):
    Mat = np.zeros((n,2*n-1))
    for i in range(n):
        for j in range(2*n-1):
            if abs(j-(n-1)) <= i:
                Mat[i,j] = 1
    return Mat
```

Exercice 3 : Triangle de Pascal

1) Très facile. On initialise la matrice à zéro et on place les 1 dans la colonne d'indice 0.

```python
import numpy as np

def Initialise(n):
    M = np.zeros((n,n))
    for i in range(n):
        M[i,0] = 1
    return M
```

2) La formule de Pascal se traduit directement sur la matrice M à construire par

$$M[i,j] = M[i-1,j] + M[i-1,j-1] \text{ (pour } (i,j) \in [\![1, m-1]\!]^2).$$

```python
def TrigDePascal(m):
    M = Initialise(m)
    for i in range(1,m):
        for j in range(1,m):
            M[i,j] = M[i-1,j] + M[i-1,j-1]
    return M
```

Exercice 4 : Relations d'amitié

Première question :

Le nombre d'amis de l'inscrit i est le nombre de 1 sur la ligne i. Comme cette ligne n'est constituée que de 1 et de 0, cela correspond aussi à la somme des éléments de la ligne i.

```python
# Exercice 4
import numpy as np

# Question 1
def nbAmis(M,n,i):
    nb = 0
    for k in range(n):
        nb += M[i,k]
    return nb
```

Deuxième question :

Il suffit d'appeler la fonction précédente sur chacune des lignes de M : on considère donc les nombres d'amis qu'ont chaque inscrit et on extrait le numéro d'un inscrit ayant le plus d'ami.

```python
# Question 2
def PlusdAmis(M):
    n = M.shape[0]
    Max = -float("inf")
    for i in range(n):
        nb = nbAmis(M,n,i)
        if nb > Max:
            Max = nb
            rg = i
    return rg
```

Troisième question :

Il suffit de compter le nombre de chiffres 1 qui se trouvent sur les lignes i et j et sur la même colonne.

```python
# Question 3
def nbAmisCommuns(M,n,i,j):
    nbAmCom = 0
    for k in range(n):
        if M[i,k] and M[j,k]:
            nbAmCom += 1
    return nbAmCom
```

Commentaire sur le code : un entier non nul considéré comme booléen vaut `True`. Seul l'entier 0 considéré comme booléen vaut `False`.

Quatrième question :

On crée une liste T initialisée à `False` dont la k-ième entrée T[k] vaudra `True` si l'inscrit k est ami d'un ami de l'inscrit i mais pas de i et T[k] = `False` sinon.

```python
# Question 4
def nbAmAmPasAm(M,n,i):
    T = [False]*n
    T[i] = True
    nb = 0
    for k in range(n):
        if M[i,k]:
            for j in range(n):
                if M[k,j] and not(M[i,j]) and not(T[j]):
                    nb += 1
                    T[j] = True
    return nb
```

Commentaires du code :

Ligne 4 : `T[i] = True` car on ne considère que les amis indirects de i, autres que i.

Ligne 7 : `if M[i,k]:` si k est ami avec i ...

Ligne 9 : `if M[k,j] and not(M[i,j]) and not(T[j]):`
...si k est ami avec j, mais j n'est pas ami avec i et si j n'a pas encore été considéré...

Ligne 10 : `nb += 1`
... alors on incrémente la variable de comptage...

Ligne 11 : `T[j] = True`
...et on prend en compte j comme ami indirect de i.

Exercice 5 : Jeu de fléchettes

1) Le tableau est carré de taille $2n-1$. Le coefficient central est sur la ligne et colonne $n-1$.
Si (i,j) sont les numéros de ligne et colonne d'un coefficient, celui vaudra
$(n-1) - \max(|i-(n-1)|, |j-(n-1)|)$. Autrement dit, on retranche à $n-1$ le plus grand écart entre l'écart de la ligne i et la ligne médiane et l'écart entre la colonne j et la colonne médiane.

```
# Question 1
def Cible(n):
    cible = np.zeros((2*n-1,2*n-1))
    for i in range(2*n-1):
        for j in range(2*n-1):
            cible[i,j] = (n-1) - max(abs(i-(n-1)),abs(j-(n-1)))
    return cible
```

2) On choisit p fois i et j au hasard dans $[\![0, 2n-2]\!]$ et on augmente le compteur du nombre de points correspondant au coefficient à la ligne i et colonne j du tableau précédent.

```
# Question 2
def SimuleTir(n,p):
    comptePoints = 0
    cible = Cible(n)
    for k in range(p):
        i, j = np.random.randint(0,2*n-1,2)
        comptePoints += cible[i,j]
    return comptePoints
```

Commentaire sur le code :
Si a, b sont des entiers et n est un entier naturel,
`np.random.randint(a,b,n)` crée une matrice ligne à n colonnes formée d'entiers choisis au hasard dans $[\![a, b-1]\!]$.

Prenez garde !
Avec le `randint` de la bibliothèque `random`, l'instruction `randint(a,b)` fournit un entier choisi au hasard dans $[\![a, b]\!]$ (ici la borne b est incluse).

Les algorithmes de tri

(pas mal de parties inspirées de http://bcpst.parc.free.fr/joomla/DOCUMENTS/Maths952/Info/TD4.pdf)

Cours sur les algorithmes de tri

Dans tout le TP, on suppose que l'on dispose d'une liste L constituée de nombres (entiers ou flottants) dont on souhaite ordonner les éléments par ordre croissant. On note n la longueur de cette liste.
Nous allons étudier trois algorithmes de tri. Mais dans ce TP, nous nous limiterons aux deux plus simples (mais aussi moins efficaces).

I - Le tri par insertion

Principe : Le tri par insertion est un algorithme assez intuitif, que la plupart des personnes utilisent naturellement pour trier des cartes (prendre des cartes mélangées une à une sur la table, et former une main en insérant chaque carte à sa place).
Au début, on ne considère que la liste constituée du seul premier élément est triée ; puis on trie les deux premiers éléments ;
ensuite, on met le troisième élément à sa place parmi les deux premiers, et ainsi de suite...
De manière générale, au moment où l'on considère le k-ième élément, les éléments qui le précèdent sont tous déjà triés. Il faut donc trouver le rang où cet élément doit être inséré en l'échangeant successivement avec l'élément qui le précède immédiatement jusqu'à trouver sa place définitive.

Exemple :
Par exemple, si L=[6,2,8,5,4,1], alors on obtient successivement (les cases en gris correspondent aux éléments déjà traités) :

6	2	8	5	4	1	
2	6	8	5	4	1	on insère le "2" à sa place
2	6	8	5	4	1	on insère le "8" à sa place
2	5	6	8	4	1	on insère le "5" à sa place
2	4	5	6	8	1	on insère le "4" à sa place
1	2	4	5	6	8	on insère le "1" à sa place, et la liste est triée

On présente ci-dessous le code du tri par insertion où la variable pos représente l'indice de l'élément en cours d'insertion.

```
def TriParIns(Liste):
    for indice in range(len(Liste)):
        pos = indice
        while pos > 0 and Liste[pos-1] > Liste[pos]:
            Liste[pos - 1], Liste[pos] = Liste[pos], Liste[pos - 1]
            pos -= 1
```

On pourra aussi se référer au site http://visualgo.net/en/sorting pour une illustration animée de cet algorithme.

II - Le tri à bulles

Le tri à bulles consiste à regarder les différentes valeurs adjacentes d'une liste (disons à n éléments), et à les permuter si le premier des deux éléments est supérieur au second. Plus précisément, les deux premiers éléments de la liste sont comparés : si le premier est supérieur au second, une permutation est effectuée. Ensuite sont comparés et éventuellement permutés les deuxième et troisième éléments,..., jusqu'aux éléments $n-1$ et n. Une fois cette étape achevée, le dernier élément de la liste est le plus grand. On recommence l'opération sur les $n-1$ éléments restants, et ainsi de suite. On s'arrête lorsqu'il n'y a plus qu'un seul élément à traiter.
Cet algorithme porte le nom de tri à bulles, car le plus grand élément remonte petit à petit vers le haut, comme une bulle !

Exemple : Si L=[6,2,8,5,4,1], alors on obtient successivement (les cases en gris clair correspondent aux éléments qui sont comparés et éventuellement échangés à l'étape suivante, et les cases en gris foncé aux éléments bien positionnés).

6	2	8	5	4	1	on échange "6" et "2".
2	6	8	5	4	1	on ne fait rien.
2	6	8	5	4	1	on échange "8" et "5".
2	6	5	8	4	1	on échange "8" et "4".
2	6	5	4	8	1	on échange "8" et "1".
2	6	5	4	1	8	après un premier parcours, le "8" est "remonté".
⋮		⋮		⋮		
2	5	4	1	6	8	après un deuxième parcours, le "6" est "remonté".
⋮		⋮		⋮		
2	4	1	5	6	8	après un troisième parcours, le "5" est "remonté".
⋮		⋮		⋮		
2	1	4	5	6	8	après un quatrième parcours, le "4" est "remonté".
⋮		⋮		⋮		
1	2	4	5	6	8	après un cinquième (et dernier) parcours, le "2" est "remonté".

On présente ci-dessous le code du tri à bulles.

```
1 def TriaBulle(L):
2     taille = len(L)
3     for tailleBis in range(taille, 0, -1):
4         for ind in range(tailleBis - 1):
5             if L[ind] > L[ind + 1]:
6                 L[ind], L[ind + 1] = L[ind + 1], L[ind]
```

On pourra aussi se référer au site http://visualgo.net/en/sorting pour une illustration animée de cet algorithme.

Exercices sur les algorithmes de tri

Exercice 1 :
Écrire une fonction `Python` adaptant et généralisant l'algorithme de tri par insertion. Cette fonction devra prendre en entrée :

- La liste à trier,
- une fonction f déterminant la relation d'ordre à appliquer sur les objets de la liste à trier. Concrètement, si a et b sont deux objets de la liste, $f(a,b)$ sera un booléen égal à `True` ssi $a < b$ au sens de l'ordre défini par f.

La fonction devra donner en sortie la liste triée suivant l'ordre défini par f. Concrètement, si a et b sont deux objets de la liste, a doit se trouver avant b dans la liste triée ssi $f(a,b)$ est égal à `True`.

Exercice 2 :
Écrire une fonction `Python` prenant en entrée un fichier texte formé de plusieurs lignes de texte. La fonction doit renvoyer en sortie un fichier texte formé des mêmes lignes mais rangées dans l'ordre croissant du nombre de caractères qui les composent. Faire de même dans l'ordre décroissant. On pourra utiliser la fonction écrite dans l'exercice précédent.
Indication Pour lire et écrire dans un fichier texte :

```
import os

# mettre le chemin absolu menant au répertoire des fichiers
os.chdir(r"C:\Users\nicolas\Desktop\DocuTextes")

# Lire les lignes du texte (stockées dans une liste L)
fichier = open('miseAjour.txt')
L = fichier.readlines()
fichier.close()

# Ecrire dans un autre fichier des lignes
# "\n" est le caractère de retour à la ligne.
M = ["ligne1: bonjour\n", "ligne2: au revoir\n"]
fichierBis = open('autreFichier.txt','w')
fichierBis.writelines(M)
fichierBis.close()
```

Exercice 3 :

Écrire une fonction `Python` qui calcule la médiane d'une liste `L` de nombres (valeur telle qu'il y ait autant d'éléments de `L` strictement inférieurs à cette valeur que d'éléments de `L` strictement supérieurs). Plus précisément,

- Dans le cas d'une liste avec un nombre impair de nombres, la médiane sera un élément de `L`.
- Dans le cas d'une liste avec un nombre pair de nombres, on prend la moyenne des 2 nombres du "milieu".

Exemple : La liste (2,4,6,7,3,5,6) devient une fois ordonnée (2,3,4,5,6,6,7). La médiane est donc 5 (car au "milieu" de la liste). La liste (1,4,1,5,1,6) devient une fois ordonnée (1,1,1,4,5,6). La médiane est donc 2.5 (moyenne de 1 et de 4).

Exercice 4 : *Distance la plus courte entre deux points.*

1. Écrire de deux façons une fonction `Python` prenant en entrée une liste L de nombres réels et donnant en sortie le plus petit nombre de la forme $|a-b|$ où a, b sont deux éléments de L (de rangs différents dans L).
   (a) La première version procèdera directement sur L.
   (b) La seconde version utilisera la liste L mais triée dans l'ordre croissant (via la méthode `sort()`).
2. Écrire une fonction `Python` prenant en entrée deux entiers naturels non nuls n, m et donnant en sortie une liste L formée de n nombres réels choisis aléatoirement dans $[0, m]$. Utiliser cette fonction (pour différentes valeurs de n, m) pour générer des listes à donner en entrée à celle de la question précédente. Commenter les résultats obtenus.

Exercice 5 :

La fonction clock() du module time permet d'enregistrer une date donnée (en seconde) à la manière d'un chronomètre. Ainsi, la fonction suivante permet de connaître le temps d'exécution (en s) de la fonction fct_tri pour trier la liste L.

```
import time
def duree(fct_tri, L):
    debut = time.clock()
    fct_tri(L)
    fin = time.clock()
    return(fin - debut)
```

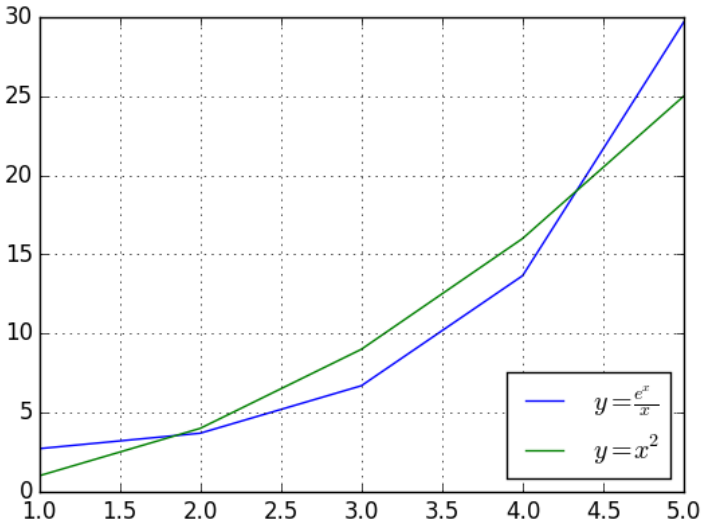
1. Ecrire une fonction liste(n) qui crée une liste de longueur n constituée d'entiers choisis aléatoirement dans $[\![0, 10n]\!]$.
2. Ecrire une fonction tps_moyen(fct_tri, n) qui crée 5 listes de longueur n à l'aide de la fonction précédente, calcule le temps nécessaire pour trier ces listes avec la fonction fct_tri et qui renvoie le temps moyen de tri obtenu.
3. Appliquer la fonction précédente aux deux fonctions de tris étudiées, et remplir le tableau suivant les différents temps moyens obtenus (en ms) :

listes de longueur	50	100	250	500	750	1000	2000	3500	5000
temps moyen du tri par insertion									
temps moyen du tri à bulles									

4. Pour une liste de longueur n, on note $T(n)$ le temps moyen en ms obtenu précédemment pour trier la liste. Tracer, pour les deux fonctions de tri, $T(n)$ en fonction de n pour $n \in \{50, 100, 250, 500, 750, 1000, 2000, 3500, 5000\}$. Commenter les résultats obtenus.

Rappel : *Exemple de construction de graphes de deux fonctions (avec légendes)*
On trace ici le graphe de $x \mapsto \frac{e^x}{x}$ sur $[1, 5]$ et de $x \mapsto x^2$ (avec légende), pour $x \in \{1, 2, 3, 4, 5\}$.

```python
from matplotlib import pyplot as plt
import numpy as np

abscisses = np.array([1,2,3,4,5])
# ou abscisses = np.array(range(1,6))
ordonnees = np.exp(abscisses)/abscisses


# Ajustement de l'axe des abscisses...
plt.axis(xlim=(0.5,5.5))

# ... avec une grille
plt.grid(True)

# ... puis tracé des courbes...
plt.plot(abscisses,ordonnees,label=r'$y=\frac{e^x}{x}$')
plt.plot(abscisses,abscisses**2,label=r'$y=x^2$')

#...avec ajout d'une légende en bas à droite
plt.legend(loc='upper right')
#... les "label" sont indispensables pour la légende.
```



Exercice 6 : *Recherche dichotomique dans une liste triée*

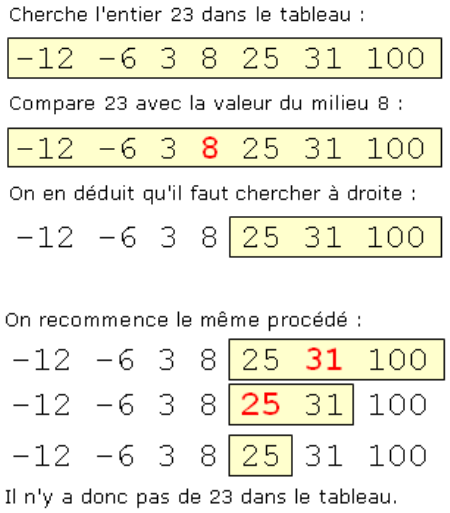
1. Écrire une fonction `Python` prenant en entrée une liste `L` de nombres entiers, un entier a, et donnant en sortie `False` si a n'est pas présent dans `L` et `(True,pos)` sinon où `pos` est un indice de la liste `L` tel que `L[pos]`$= a$.

2. Écrire une autre version de la fonction précédente, elle devra trier la liste L (par la méthode `sort()`) puis traduire le pseudo-code ci-dessous (recherche dichotomique dans une liste triée) :

```
gauche=0
droite=TailleL - 1
Tant que (droite-gauche > 0)
 milieu = (gauche + droite)/2 (division entière)
 Si a > L[milieu]
   gauche = milieu + 1
 sinon
   droite = milieu
Si L[gauche] = a
 renvoyer (True,gauche)
sinon
 renvoyer False
```

Explications :

Pour chercher un nombre entre les indices `gauche` et `droite` d'un tableau trié, on regarde la valeur se trouvant exactement à l'indice milieu de `gauche` et `droite`, et on la compare à la valeur que l'on cherche, on en déduit s'il faut continuer à chercher sur la moitié gauche ou sur la moitié droite. C'est un principe que l'on appelle la dichotomie. Voici un exemple d'exécution :



3. Écrire une fonction liste(n) qui crée une liste de longueur n constituée d'entiers choisis aléatoirement dans $[\![0, 2n]\!]$.

4. Pour une liste L de longueur n et un entier a, on note $T(n)$ le temps en ms nécessaire pour déterminer si a appartient ou non à la liste. Tracer $T(n)$ en fonction de n pour les deux versions des questions 1) et 2) où $n \in \{1000, 5000, 10000, 100000\}$. On générera les listes à l'aide de la fonction écrite précédemment et a pourra être choisi aléatoirement dans $[\![0, 2n]\!]$ (mais prendre le même a pour les deux fonctions). Répéter plusieurs fois cette représentation graphique et commenter.

Exercice 7 : *Temps de travail (exercice emprunté au site http://www.france-ioi.org/)*

Une tâche automatisé par une machine doit être effectuée de la manière la plus rentable possible. Le problème est que pour assurer le bon fonctionnement de la machine, une personne doit toujours être postée à coté de la machine. Vous êtes patron et vous disposez des différentes disponibilités de vos employés (chacun a indiqué sa disponibilité sous forme d'intervalle de temps entre deux dates, ces dates étant exprimées en nombre de secondes depuis le 1^{er} janvier). Une seule personne peut travailler à ce poste à un moment donné. Vous devez déterminer le temps total maximal d'utilisation de la machine qu'il est possible d'accumuler d'après les disponibilités que l'on vous a fournies.

Ecrire une fonction `Python` qui :

- prend en entrée une liste de couples, avec autant de couples que d'employés. Chacun des couples contient deux entiers : les bornes de début et de fin d'un intervalle de disponibilité de l'employé.
- retourne en sortie le nombre total de secondes pendant lesquelles le poste peut être occupé.

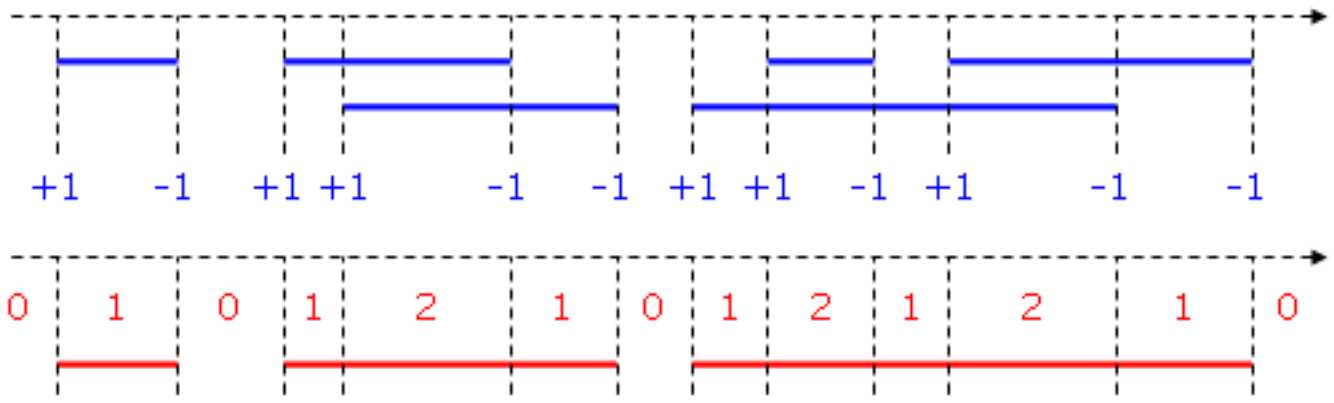
Exemple :
En entrée : $[(1,3),(7,15),(20,22),(2,5),(9,11),(12,18)]$ En sortie : 17

Commentaires :

Le poste peut être occupé de la seconde 1 à la seconde 5, puis de la seconde 7 à la seconde 18, et finalement de la seconde 20 à la seconde 22, soit un total de $4 + 11 + 2 = 17$ secondes

Principe de la démarche à suivre :

- Affecter un code à chaque date : 1 pour le début d'une disponibilité, -1 pour la fin d'une disponibilité.
- Créer et trier les couples (dates, code) par dates croissantes.
- Lorsqu'on passe les évènements (dates, code), on maintient un compteur qui nous dit combien de segments de disponibilité sont ouverts à l'instant courant. S'il y a au moins un segment ouvert, alors la taille de l'intervalle entre les deux évènements successifs considérés est à ajouter au total.



Corrigés des exercices sur le tri

Exercice 1 :

Il suffit simplement de reprendre les lignes 1 et 4 du code du tri par insertion. On remplace `Liste[pos]<Liste[pos-1]` par `f(Liste[pos],Liste[pos-1])`. On en profite aussi pour définir une variable de sortie.

```
1  def TriParIns(Liste, f):
2      for indice in range(len(Liste)):
3          pos = indice
4          while pos > 0 and f(Liste[pos],Liste[pos-1]):
5              Liste[pos - 1], Liste[pos] = Liste[pos], Liste[pos - 1]
6              pos -= 1
7      return Liste
```

Illustration : supposons donnée une liste L de chaînes de caractères. Chaque chaîne comporte un certain nombre de "e" ou de "E". On veut classer ces chaînes par nombre de "e" (ou "E") croissant.
On commence par écrire notre fonction de comparaison f.

```
1  def f(a,b):
2      nbeDansa = 0
3      for elem in a:
4          if elem == "e" or elem == "E":
5              nbeDansa += 1
6      nbeDansb = 0
7      for elem in b:
8          if elem == "e" or elem == "E":
9              nbeDansb += 1
10     return nbeDansa < nbeDansb
```

Explications :
On compte le nombre de "e" ou "E" dans la chaîne `a` et dans la chaîne `b`. On donne en sortie l'expression booléenne `nbeDansa` $<$ `nbeDansb` qui vaut `True` si `nbeDansa` est strictement inférieur à `nbeDansb` et qui vaut `False` sinon.

On fait alors appel à la fonction précédente sur un exemple de liste.

```
1  L=["Rouge", "et vous, que faites-vous?", "chat gris", "Euripide"]
2  TriParIns(L,f)
3  >>> L
4  >>> ['chat gris', 'Rouge', 'Euripide', 'et vous, que faites-vous?']
```

Exercice 2 :

Dans le fichier texte `mon_Texte.txt`, on tapera sur `Return` à la fin de la dernière ligne avant de sauvegarder le fichier (le curseur se placera alors sur la première ligne non remplie). En effet, à la fin de chaque ligne du texte se trouve un caractère invisible de "retour à la ligne" noté `\n`. Toutes les lignes en sont pourvus sauf la dernière si on n'effectue pas la petite manipulation indiquée ci-dessus.

Version 1 : on ne fait pas appel à la fonction de l'exercice 1 et on procède directement sachant que `len(C)` est le nombre de caractères d'une chaîne `C`.

```python
import os

# tri par longueurs croissantes
def Tri1(fichier):
    Liste = fichier.readlines()
    fichier.close()
    # on trie L (par insertion)
    for indice in range(len(Liste)):
        pos = indice
        while pos > 0 and len(Liste[pos-1]) > len(Liste[pos]):
            Liste[pos - 1], Liste[pos] = Liste[pos], Liste[pos - 1]
            pos -= 1
    fichierBis = open('mon_Texte_trié_c.txt','w')
    fichierBis.writelines(Liste)
    return Liste

# tri par longueurs décroissantes: on remplace juste le > par <.
def Tri2(fichier):
    Liste = fichier.readlines()
    fichier.close()
    # on trie L (par insertion)
    for indice in range(len(Liste)):
        pos = indice
        while pos > 0 and len(Liste[pos-1]) < len(Liste[pos]):
            Liste[pos - 1], Liste[pos] = Liste[pos], Liste[pos - 1]
            pos -= 1
    fichierBis = open('mon_Texte_trié_d.txt','w')
    fichierBis.writelines(Liste)
    return Liste

""" exécution des deux fonctions """

# mettre le chemin absolu menant au répertoire des fichiers (en préfixant par la lettre r)
os.chdir(r"C:\Users\nicolas\Desktop")

# Lire les lignes du texte (stockées dans une liste L)
fichier = open('mon_Texte.txt')
Tri1(fichier)
fichier = open('mon_Texte.txt')
Tri2(fichier)
```

Version 2 : on fait appel à la fonction précédente avec deux relations de comparaison personnalisées.

```python
# version 2
# comp1 renvoie True ssi len(a) < len(b)
def comp1(a,b):
    return len(a) < len(b)

def comp2(a,b):
    return len(a) > len(b)

def TriParIns(Liste, f):
    for indice in range(len(Liste)):
        pos = indice
        while pos > 0 and f(Liste[pos],Liste[pos-1]):
            Liste[pos - 1], Liste[pos] = Liste[pos], Liste[pos - 1]
            pos -= 1
    return Liste

def TriTexte(fichier, comp):
    Liste = fichier.readlines()
    fichier.close()
    # on trie L (par insertion)
    TriParIns(Liste, comp)
    fichierBis = open('mon_Texte_encore_trié.txt','w')
    fichierBis.writelines(Liste)
    return Liste

""" exécutions """

# mettre le chemin absolu menant au répertoire des fichiers (en préfixant par la lettre r)
os.chdir(r"C:\Users\nicolas\Desktop")

# Lire les lignes du texte (stockées dans une liste L)
fichier = open('mon_Texte.txt')
TriTexte(fichier, comp1)
fichier = open('mon_Texte.txt')
TriTexte(fichier, comp2)
```

Exercice 3 :

```python
# Exercice 3
def Mediane(L):
    L.sort()
    if len(L)%2:  # S'il y a un nombre impair d'éléments
        return L[len(L)//2]
    else: # s'il y a un nombre pair d'éléments
        return (L[len(L)//2-1] + L[len(L)//2])/2
```

Exercice 4 : *Distance la plus courte entre deux points.*

1)

```python
# Exercice 4, question 1
# 1) a)
def distPlusCourte(L):
    Min = float("inf")
    for i in range(len(L)):
        for j in range(i+1,len(L)):
            if abs(L[i]-L[j]) < Min:
                Min = abs(L[i]-L[j])
    return Min

# 1) b)
def distPlusCourteBis(L):
    Min = float("inf")
    L.sort()
    for i in range(1, len(L)):
        if L[i]-L[i-1] < Min:
            Min = L[i]-L[i-1]
    return Min
```

2)

```python
# Exercice 4, question 2
from random import randint

def Liste(n,m):
    return [randint(0,m) for k in range(n)]
```

En faisant quelques tests, on trouve évidemment les mêmes résultats pour les deux fonctions.

Mais ce qui aurait été intéressant : comparer les temps d'exécution des deux fonctions. On aurait remarqué que le temps de tri de la liste L par la deuxième fonction est négligeable devant le temps d'exécution des deux boucles `for` imbriquées de la première. Ainsi, la deuxième fonction (celle du 1) b)) est plus rapide.

Exercice 5 :
On écrit d'abord les fonctions utiles.

```
# tri par insertion
def TriParIns(Liste):
    for indice in range(len(Liste)):
        pos = indice
        while pos > 0 and Liste[pos-1] > Liste[pos]:
            Liste[pos - 1], Liste[pos] = Liste[pos], Liste[pos - 1]
            pos -= 1

# tri à bulles
def TriaBulle(L):
    taille = len(L)
    for tailleBis in range(taille, 0, -1):
        for ind in range(tailleBis - 1):
            if L[ind] > L[ind + 1]:
                L[ind], L[ind + 1] = L[ind + 1], L[ind]


import time, random

def duree(fct_tri, L):
    debut = time.clock()
    fct_tri(L)
    fin = time.clock()
    return(fin - debut)
```

1)

```
# Question 1
def liste(n):
    return [random.randint(0,10*n) for k in range(n)]
```

2)

```python
# Question 2
def tps_moyen(fct_tri, n):
    temps = 0
    for k in range(5):
        L = liste(n)
        temps += duree(fct_tri, L)
    return temps/5
```

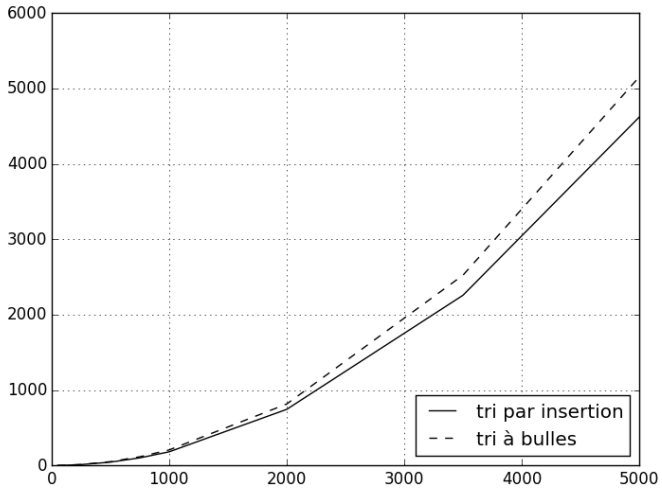
3)

```python
# Question 3
""" on va créer un tableau numpy à deux lignes:
la première pour le tri par insertion,
la seconde pour le tri à bulles."""
import numpy as np

def ListesTpsMoyens():
    longueurs = [50,100,250,500,750,1000,2000,3500,5000]
    Tps1 = [tps_moyen(TriParIns,long)*1000 for long in longueurs]
    Tps2 = [tps_moyen(TriaBulle,long)*1000 for long in longueurs]
    return np.array([Tps1,Tps2])
```

4)

```python
# Question 4
from matplotlib import pyplot as plt

TpsMoyens = ListesTpsMoyens()
abscisses = [50,100,250,500,750,1000,2000,3500,5000]
ordonnees1 = np.array(TpsMoyens[0,:])
ordonnees2 = np.array(TpsMoyens[1,:])

# Ajustement de l'axe des abscisses...
plt.axis(xlim=(50,5000))
# ... avec une grille
plt.grid(True)

# ... puis tracé des courbes en noir...
plt.plot(abscisses,ordonnees1,label='tri par insertion',c='black')
plt.plot(abscisses,ordonnees2,ls = '--',label='tri à bulles',c='black')

#...avec ajout d'une légende en bas à droite
plt.legend(loc='lower right')
#... les "label" sont indispensables pour la légende.
```

Voilà le type de graphique que l'on obtient.



On remarque que les temps de tris sont à peu près semblables avec un léger mieux pour le tri par insertion. Le temps de tri d'une liste de 5000 éléments est de l'ordre de 5s pour les deux algorithmes, ce qui est assez lent.

On remarque également que les deux courbes ressemblent à des portions de paraboles (en fait, on montre facilement que ces deux algorithmes de tris nécessitent de l'ordre de n^2 opérations pour une liste de longueur n, ce qui conforte cette dernière remarque).

Exercice 6 :

1)

```python
# Exercice 6, question 1
def Cherche(L,a):
    for pos in range(len(L)):
        if L[pos] == a:
            return (True, pos)
    return False
```

2)

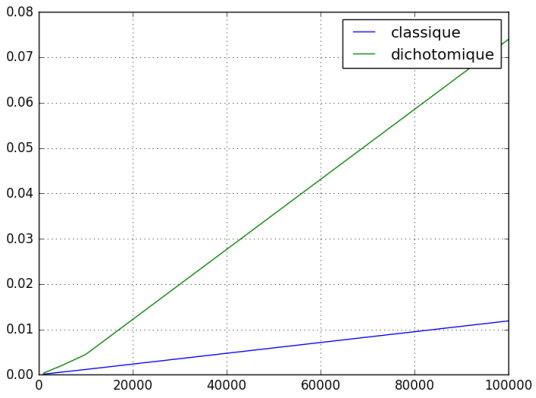
```python
# Exercice 6, question 2
def ChercheBis(L,a):
    L.sort()
    gauche = 0
    droite = len(L)-1
    while droite - gauche > 0:
        milieu = (gauche + droite)//2
        if a > L[milieu]:
            gauche = milieu + 1
        else:
            droite = milieu
    if L[gauche] == a:
        return (True, gauche)
    else:
        return False
```

3)

```python
# Exercice 6, question 3
from random import randint

def liste(n):
    return [randint(0,2*n) for k in range(n)]
```

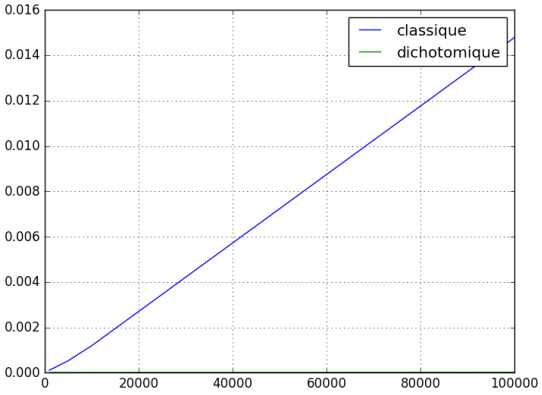
4)

```python
# Exercice 6, question 4
import time
from matplotlib import pyplot as plt

def Graphiques(a):
    Liste_n = [1000,5000,10**4,10**5]
    ordonnees1, ordonnees2 = [], []
    for n in Liste_n:
        L = liste(n)
        top1 = time.clock()
        Cherche(L,a)
        top2 = time.clock()
        ChercheBis(L,a)
        top3 = time.clock()
        ordonnees1.append(top2-top1)
        ordonnees2.append(top3-top2)
    plt.axis(xlim=(999,10**5+1))
    plt.grid(True)
    plt.plot(Liste_n, ordonnees1, label=r'classique')
    plt.plot(Liste_n, ordonnees2, label=r'dichotomique')
    plt.legend(loc='upper right')
    plt.show()
```



Le temps d'exécution de la fonction du 2) est plus long à cause du temps du tri de la liste.
En revanche, pour une liste de départ est déjà triée, la fonction du 2) (si elle ne trie pas à nouveau) est bien plus rapide !

```python
def ChercheBibis(L,a):
    gauche = 0
    droite = len(L)-1
    while droite - gauche > 0:
        milieu = (gauche + droite)//2
        if a > L[milieu]:
            gauche = milieu + 1
        else:
            droite = milieu
    if L[gauche] == a:
        return (True, gauche)
    else:
        return False

def GraphiquesBis(a):
    Liste_n = [1000,5000,10**4,10**5]
    ordonnees1, ordonnees2 = [], []
    for n in Liste_n:
        L = liste(n)
        L.sort()
        top1 = time.clock()
        Cherche(L,a)
        top2 = time.clock()
        ChercheBibis(L,a)
        top3 = time.clock()
        ordonnees1.append(top2-top1)
        ordonnees2.append(top3-top2)
    plt.axis(xlim=(999,10**5+1))
    plt.grid(True)
    plt.plot(Liste_n, ordonnees1, label=r'classique')
    plt.plot(Liste_n, ordonnees2, label=r'dichotomique')
    plt.legend(loc='upper right')
    plt.show()
```



Exercice 7 :

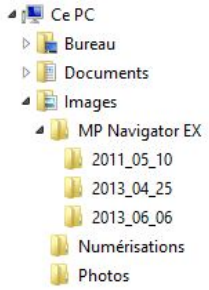
```python
# Exercice 7
def TempsTravail(L):
    # Création de la liste des (date, code):
    Liste = []
    for couple in L:
        debut, fin = couple
        Liste.append((debut,1))
        Liste.append((fin,-1))
    # Tri de cette dernière liste par dates croissantes:
    for indice in range(len(Liste)):
        pos = indice
        while pos > 0 and Liste[pos-1][0] > Liste[pos][0]:
            Liste[pos-1], Liste[pos] = Liste[pos], Liste[pos-1]
            pos -= 1
    # Calcul du temps de travail cumulé:
    tempsTravail = 0
    compteur = Liste[0][1]
    for k in range(1, len(Liste)):
        if compteur > 0:
            tempsTravail += Liste[k][0] - Liste[k-1][0]
        compteur += Liste[k][1]
    return tempsTravail
```

Récursivité et tri rapide

Cours sur la récursivité et le tri rapide

1) Introduction (introduction inspirée d'un cours de Mathias Hiron présent sur l'ancienne version du site http://www.france-ioi.org/)

Une poupée russe est un objet en forme de personnage, qui est vide ou qui contient une poupée russe semblable, mais plus petite. Pour décrire ce qu'est une poupée russe, on a utilisé le nom de l'objet lui-même. Cette définition est pourtant très claire, même pour quelqu'un qui n'a jamais vu un tel objet. On dit qu'on a utilisé une définition récursive d'une poupée russe.



Un répertoire est un élément informatique qui est vide ou qui contient des fichiers ou des répertoires. Sauriez-vous définir la notion de répertoire sans réutiliser le mot répertoire, et sans rendre la définition incomplète ? Pas facile... un répertoire est en lui-même un concept récursif. On peut remarquer plusieurs points communs entre nos deux exemples, qui décrivent pourtant des choses très différentes : l'un est un objet du monde réel, tandis que l'autre n'est qu'une abstraction, qui n'existe que d'un point de vue logiciel.

Autre exemple, plus mathématique celui-ci, si n est un entier naturel, le nombre $n!$, c'est 1 si $n = 0$, et c'est $n \times (n-1)!$ sinon. On a bien défini encore $n!$ de manière récursive.

Deux principes fondamentaux communs à nos trois définitions se dégagent alors :

1. On a une condition d'arrêt (la poupée russe est vide, le répertoire est vide, $n! = 1$ si $n = 0$).
2. On a le coeur de la définition récursive (si la condition d'arrêt n'est pas valide, une poupée russe contient une autre poupée russe, un répertoire peut contenir un autre répertoire, $n!$ c'est n multiplié par $(n-1)!$).

2) Exemples de fonctions récursives avec simple appel

Nous allons examiner ci-dessous des exemples de fonctions dont le corps fait un simple appel à cette même fonction.

En remarquant que $N! = 1 \times 2 \times 3 \times \cdots \times N$ (avec $0! = 1$ suivant la convention du produit vide), il est possible d'écrire une fonction factorielle(N) (qui calcule $N!$ si N est un entier naturel) n'utilisant qu'une simple boucle for :

```
1 def factorielle(N):
2         Fact = 1
3         for k in range(2, N+1):
4                 Fact *= k
5         return Fact
```

Il est cependant possible de donner une définition récursive de la fonction factorielle comme nous l'avons vu plus haut.

La factorielle d'un nombre N vaut 1 si N est égal à 0, et N multiplié par la factorielle de $N-1$ sinon.

Cette définition est parfaitement équivalente à la précédente, et peut se traduire en code par une fonction récursive (plus simple que la précédente) :

```
def factorielleBis(N):
        if N == 0:
                return 1
        return N*factorielleBis(N-1)
```

La première version, qui utilise une boucle, est ce que l'on appelle une implémentation *itérative* de la fonction factorielle : on effectue un certain nombre d'itérations d'une boucle. La deuxième version s'appelle tout simplement l'implémentation *récursive*.

Illustration des appels récursifs de la fonction `factorielleBis` appelée avec $n = 4$ en entrée :

$4 \neq 0$
appel de `factorielleBis(3)`
$3 \neq 0$
appel de `factorielleBis(2)`
$2 \neq 0$
appel de `factorielleBis(1)`
$1 \neq 0$
appel de `factorielleBis(0)`
$0 = 0$
valeur de sortie : 1
Évaluation de 1×1
valeur de sortie : 1
Évaluation de 2×1
valeur de sortie : 2
Évaluation de 3×2
valeur de sortie : 6
Évaluation de 4×6
valeur de sortie : 24

Terminons par un point important :

Que se passerait-il si l'on avait omis dans la fonction précédente la condition d'arrêt :

```
if N == 0:
    return 1
```

La fonction serait alors :

```
def factorielleBis(N):
    return N*factorielleBis(N-1)
```

Lors de son appel, la fonction factorielleBis(N) multiplierait N par le résultat de l'appel factorielleBis(N-1) qui multiplierait elle-même $N-1$ par le résultat de l'appel factorielleBis(N-2), etc. ... cela n'aurait pas de fin, et à aucun moment, le résultat attendu ne serait affiché.

De la même manière qu'une poupée russe, pour être un objet concret, doit pouvoir ne pas toujours contenir d'autre poupée russe, une fonction récursive ne doit pas toujours se rappeler, sinon elle ne se terminera jamais. On peut ainsi définir la règle suivante :
Une fonction récursive doit toujours comporter une condition de fin des appels (condition d'arrêt), pour ne pas avoir une exécution infinie.

Donnons un autre exemple de fonction récursive :
Ecrivons un programme qui lit deux entiers, et qui affiche le premier entier, entouré d'autant de paires de crochets [et], qu'indiqué par la valeur du deuxième nombre.

Exemple 1 :

en entrée ...
42
3

,

en sortie ...
[[[42]]]

Exemple 2 :

en entrée ...
24
0

,

en sortie ...
24

On pourrait bien sûr utiliser deux boucles for pour afficher les crochets ouvrants [et fermants], mais nous allons utiliser un code plus court, basé sur une fonction récursive.
L'idée est la suivante : Un nombre encadré de N paires de crochets est ce même nombre encadré de $N-1$ paire de crochets, encadré d'une paire de crochets. Cela donne la fonction ci-dessous dont vous devez mettre la condition de fin des appels en évidence en gras :

```
1 def NombreEncadre(nombre, NbPaires):
2         if NbPaires == 0:
3                 return str(nombre)
4         return "[" + NombreEncadre(nombre, NbPaires-1) + "]"
```

Exercice 1 :
Ecrire une fonction récursive qui, à partir d'un entier naturel N, calcule la somme $\sum_{k=0}^{N} k^5$ des N premiers entiers naturels.

Exercice 2 :
Ecrire une fonction Python qui prend en paramètres d'entrée une chaîne de caractères. La fonction doit afficher en retour la chaîne "retournée",
c'est-à-dire commençant par la dernière lettre et terminant par la première. La contrainte est que cette fonction doit être récursive.

Corrigé :
On donne la formulation récursive :
Une chaîne retournée est formée du dernier caractère de cette chaîne, suivie de cette même chaîne privée de son dernier caractère puis retournée.

```python
def Retournement(chaine):
    if len(chaine) == 0:
        return ""   # chaine vide
    return chaine[-1] + Retournement(chaine[:-1])
```

Exercice 3 :

Méthode de dichotomie :

On suppose seulement f continue sur un intervalle $[a,b]$ où l'on sait que f s'annule.

On rappelle que la méthode de dichotomie consiste à construire deux suites adjacentes (a_n) et (b_n) telles que :

- $a_0 = a$, $b_0 = b$,
- $x_0 \in [a_n, b_n]$ pour tout entier naturel n,
- $b_{n+1} - a_{n+1} = \frac{1}{2}(b_n - a_n)$ (la longueur de $[a_{n+1}, b_{n+1}]$ est la moitié de celle de $[a_n, b_n]$).

On a alors $|a_n - x_0| \leqslant \frac{|b-a|}{2^n}$ pour tout entier naturel. Si n est tel que $\frac{|b-a|}{2^n} \leqslant$ epsilon, le réel a_n (de même que le point milieu $\frac{a_n+b_n}{2}$) est ainsi une approximation de x_0 à epsilon près.

Remarques

En général, on choisit $[a,b]$ de sorte que le segment ne possède qu'une seule solution x_0 de l'équation $f(x) = 0$. Les points forts de la méthode de dichotomie sont son domaine d'application assez large et la possibilité de choisir la précision de l'approximation souhaitée.

On rappelle qu'une version itérative de la méthode de dichotomie est fournie par la fonction `Python` ci-dessous

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

On demande dans cet exercice d'écrire une version **récursive** de cette fonction.

Exercice 4 :
Écrire une fonction Python récursive prenant en entrée deux nombres entiers naturels n et p.
La fonction doit afficher $2p-1$ fois le nombre n sur $2p-1$ lignes différentes en respectant (et généralisant) la mise en forme des exemples ci-dessous :

Exemples :

Si $p=3$, $n=5$. On doit afficher en sortie

```
    5
  5 5 5
5 5 5 5 5
```

. Si $p=4$ et $n=2$, on affiche

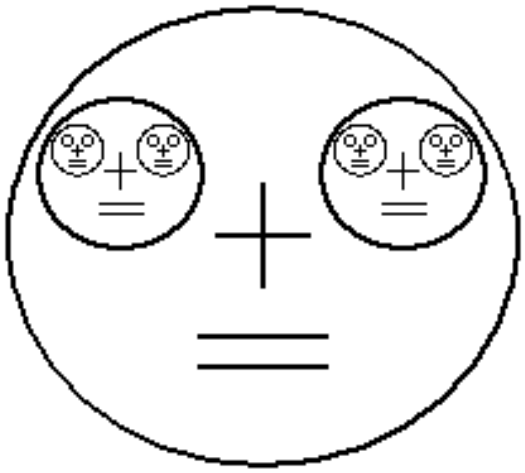
```
      2
    2 2 2
  2 2 2 2 2
2 2 2 2 2 2 2
```

.

3) Exemple d'une fonction récursive avec plusieurs appels

Le corps d'une fonction récursive peut très bien faire **plusieurs** appels à elle-même (deux ou plus).

Exercice 5 : (0 + 0 = la tête à Toto, exercice emprunté à http://www.france-ioi.org/)



Comme on le sait, 0 + 0 = 0. On pourrait aussi écrire 0 = (0 + 0). Dans ce cas, on peut aussi aller un peu plus loin, et puisque 0 vaut (0 + 0), remplacer les 0 de (0 + 0) par leur valeur, et obtenir :
0 = ((0 + 0) + (0 + 0))
Rien n'empêche de continuer et d'écrire : 0 = (((0 + 0) + (0 + 0)) + ((0 + 0) + (0 + 0)))
Ecrire une fonction Python qui, à partir d'un entier N, indique la valeur de 0, en ayant remplacé N fois les zéros à droite de l'égalité "0 = 0" par leur valeur "(0+0)".

Exemples :

en entrée ...	en sortie ...	en entrée ...	en sortie ...
0	0 = 0	2	0 = ((0 + 0) + (0 + 0))

en entrée ...	en sortie ...
3	0 = (((0 + 0) + (0 + 0)) + ((0 + 0) + (0 + 0)))

Corrigé :
Pour obtenir un code simple, il faut commencer par exprimer le problème plus simplement.
On considère que pour un entier N donné, on doit afficher le texte "0 = ", puis ce que l'on va appeler "une tête à toto de niveau N".
On peut alors donner une définition récursive de ce qu'est une tête à toto de niveau N :

❶ Cas de base : une tête à toto de niveau de niveau 0 est un 0.

❷ Cas général : une tête à toto de niveau N est composée d'une parenthèse ouvrante, une tête à toto de niveau N-1, d'un espace, un signe +, un espace, d'une autre tête à toto de niveau N-1, puis une parenthèse fermante.

On peut aussi dire qu'elle a la forme "(X + X)", où X est une tête à toto de niveau N-1.

En se basant directement sur cette définition, on obtient le pseudo-code suivant, qui utilise une fonction récursive ayant un paramètre, le "niveau" de la tête à toto.
fonction teteAToto(niveau) :
    Si niveau = 0
        Renvoyer "0"
        Quitter la fonction
    Renvoyer ("(" *suivi de* teteAToto(niveau - 1) *suivi de* " + " *suivi de* teteAToto(niveau - 1) *suivi de* ")"
Afficher "0 = " *suivi de* teteAToto(niveau)

```python
def TeteToto(niveau):
    if niveau == 0:
        return "0"
    return "("+TeteToto(niveau-1)+"+"+TeteToto(niveau-1)+")"

# un exemple d'appel de la fonction:
print("0="+TeteToto(5))
```

L'exercice suivant et son corrigé sont à retenir : **ils font partie intégrante du cours**.
Exercice 6 : **Le tri rapide (Quicksort)**
Le tri rapide (ou Quicksort en anglais) est un algorithme de tri basé sur le principe suivant : si `L` est la liste à trier (par ordre croissant), on choisit aléatoirement un élément `a` de `L` (appelé "pivot") et l'on place à gauche de cet élément tous ceux qui lui sont inférieurs, et à sa droite tous ceux qui lui sont supérieurs (opération dite de "partitionnement"). Cela divise `L` en deux sous-listes (les éléments de `L` à gauche de `a`, et ceux à sa droite) auxquelles on applique le même principe de partitionnement (si les sous-listes ont au moins 2 éléments) et ainsi de suite jusqu'à ce que `L` soit parfaitement triée.
Voyons cela sur l'exemple où `L = [0,7,-3,2,9,11,5,8,-4]` (les pivots apparaissent successivement en gris) :

0	7	−3	2	9	11	5	8	−4	choix du pivot
−4	0	−3	2	9	11	5	8	7	partitionnement
−4	0	−3	2	9	11	5	8	7	choix de deux nouveaux pivots
−4	−3	0	2	5	7	8	11	9	partitionnements
−4	−3	0	2	5	7	8	11	9	choix de 3 nouveaux pivots
−4	−3	0	2	5	7	8	9	11	partitionnements
−4	−3	0	2	5	7	8	9	11	la liste est triée

On suppose que `L` est une liste déjà définie, de taille n.
1) Écrire une fonction Python `Partitionne` prenant en entrée deux entiers p, q tels que $0 \leqslant p < q < n$ et qui

- choisit aléatoirement un élément `a` de `L` (appelé "pivot"), situé entre les indices p et q (inclus),
- pour chaque élément de `L` situé entre les indices p et q (inclus), place les éléments inférieurs ou égaux à `a` avant `a`, et les éléments supérieurs à `a` après `a` (l'élément `a` étant situé entre ces deux parties),
- La fonction ne renvoie rien en sortie (mais a modifié tout de même la liste `L`).

L'opération réalisée par cette fonction est ce qu'on appelle un partitionnement.
On impose que le partitionnement s'effectue de la manière suivante :

- On permute le pivot `a` avec l'élément d'indice p de `L` et on note *pos* la position du pivot (initialement, c'est p).
- Pour chaque élément `b` de `L` à droite de `L[pos]` (mais d'indice $\leqslant q$), si `b` est inférieur à `a`, on place `b` à la place de `L[pos]` et on déplace `L[pos]` à la position $pos + 1$ de sorte que les autres éléments à droite du pivot `L[pos]` restent à sa droite (par deux échanges dans la liste `L`) et on met à jour la variable *pos*.

2) Compléter la fonction Python précédente de sorte qu'elle soit récursive et s'appelle sur les deux sous-listes de part et d'autre du pivot choisi `a`.
Pour trier `L`, on appellera la fonction `Partitionne` avec les paramètres d'entrée 0 et $n - 1$.
On crée ainsi une fonction de tri bien plus efficace (au point de vue temps d'exécution) que le tri par insertion ou le tri à bulles (utile si la liste `L` est formée de très nombreux éléments).
On pourra également se reporter à http://visualgo.net/en/sorting (choisir R-QUICK) pour une illustration de cet algorithme de tri.

Corrigés d'exercices sur la récursivité et le tri rapide

Exercice 4 :
Écrire une fonction Python récursive prenant en entrée deux nombres entiers naturels n et p.
La fonction doit afficher $2p-1$ fois le nombre n sur $2p-1$ lignes différentes en respectant (et généralisant) la mise en forme des exemples ci-dessous :

Exemples :

Si $p = 3$, $n = 5$. On doit afficher en sortie

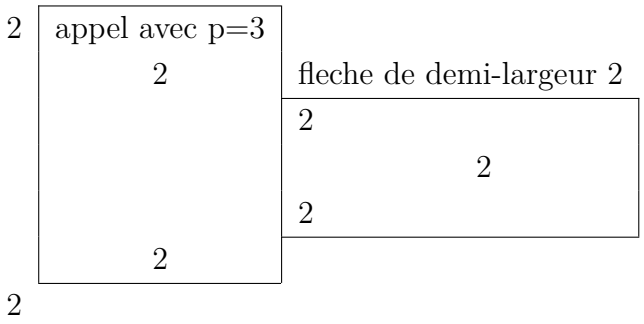
```
        2
    5       2
        5       2
    5       2
        2
```

. Si $p = 4$ et $n = 2$, on affiche

```
    2
2       2
    2
```

.

Corrigé :
Pour simplifier, on appellera "flèche de nombres n de demi-largeur p" la forme à afficher.
La formulation récursive de la fonction est alors la suivante :
Condition d'arrêt : si $p = 1$, afficher le nombre n précédé du bon nombre de tabulations.
Sinon (si $p \geqslant 2$), afficher le nombre n précédé du bon nombre de tabulations, augmenter le nombre de tabulation d'une unité, afficher une flèche de nombres n de demi-largeur $p - 1$, afficher de nouveau le nombre n précédé du même nombre de tabulations que précédemment.
Illustration :



On créera donc une variable globale `nbIndent` qui va déterminer le nombre de tabulations qui doivent précéder le nombre n à afficher sur une ligne.

```python
nbIndent = 0
def Fleche(n,p):
    global nbIndent
    if p==1:
        print("\t"*nbIndent + str(n))
        return
    print("\t"*nbIndent + str(n))
    nbIndent += 1
    Fleche(n,p-1)
    nbIndent -= 1
    print("\t"*nbIndent + str(n))
```

Exercice 6 : **Le tri rapide (Quicksort)**

Le tri rapide (ou Quicksort en anglais) est un algorithme de tri basé sur le principe suivant : si `L` est la liste à trier (par ordre croissant), on choisit aléatoirement un élément `a` de `L` (appelé "pivot") et l'on place à gauche de cet élément tous ceux qui lui sont inférieurs, et à sa droite tous ceux qui lui sont supérieurs (opération dite de "partitionnement"). Cela divise `L` en deux sous-listes (les éléments de `L` à gauche de `a`, et ceux à sa droite) auxquelles on applique le même principe de partitionnement (si les sous-listes ont au moins 2 éléments) et ainsi de suite jusqu'à ce que `L` soit parfaitement triée.
Voyons cela sur l'exemple où `L = [0,7,-3,2,9,11,5,8,-4]` (les pivots apparaissent successivement en gris) :

0	7	−3	2	9	11	5	8	−4	choix du pivot
−4	0	−3	2	9	11	5	8	7	partitionnement
−4	0	−3	2	9	11	5	8	7	choix de deux nouveaux pivots
−4	−3	0	2	5	7	8	11	9	partitionnements
−4	−3	0	2	5	7	8	11	9	choix de 3 nouveaux pivots
−4	−3	0	2	5	7	8	9	11	partitionnements
−4	−3	0	2	5	7	8	9	11	la liste est triée

On suppose que `L` est une liste déjà définie, de taille n.
1) Écrire une fonction Python `Partitionne` prenant en entrée deux entiers p, q tels que $0 \leqslant p < q < n$ et qui

- choisit aléatoirement un élément `a` de `L` (appelé "pivot"), situé entre les indices p et q (inclus),
- pour chaque élément de `L` situé entre les indices p et q (inclus), place les éléments inférieurs ou égaux à `a` avant `a`, et les éléments supérieurs à `a` après `a` (l'élément `a` étant situé entre ces deux parties),
- La fonction ne renvoie rien en sortie (mais a modifié tout de même la liste `L`).

L'opération réalisée par cette fonction est ce qu'on appelle un partitionnement.
On impose que le partitionnement s'effectue de la manière suivante :

- On permute le pivot `a` avec l'élément d'indice p de `L` et on note *pos* la position du pivot (initialement, c'est p).
- Pour chaque élément `b` de `L` à droite de `L[pos]` (mais d'indice $\leqslant q$), si `b` est inférieur à `a`, on place `b` à la place de `L[pos]` et on déplace `L[pos]` à la position $pos + 1$ de sorte que les autres éléments à droite du pivot `L[pos]` restent à sa droite (par deux échanges dans la liste `L`) et on met à jour la variable *pos*.

2) Compléter la fonction Python précédente de sorte qu'elle soit récursive et s'appelle sur les deux sous-listes de part et d'autre du pivot choisi `a`.
Pour trier `L`, on appellera la fonction `Partitionne` avec les paramètres d'entrée 0 et $n - 1$.
On crée ainsi une fonction de tri bien plus efficace (au point de vue temps d'exécution) que le tri par insertion ou le tri à bulles (utile si la liste `L` est formée de très nombreux éléments).
On pourra également se reporter à http://visualgo.net/en/sorting (choisir R-QUICK) pour une illustration de cet algorithme de tri.

Corrigé :

1) On choisit au hasard un pivot qu'on place en position p. On regarde chaque élément à droite du pivot (représenté par `L[u]` dans le code ci-dessous), s'il est inférieur au pivot, on le permute avec celui-ci, puis on permute le pivot avec l'élément qui le suivait immédiatement.

```
from random import *

def Partitionne(p,q):
    pos = randint(p,q)
    a = L[pos]
    L[p], L[pos] = L[pos], L[p]
    pos, u = p, p+1
    while u <= q:
        if L[u] < a and pos < q:
            L[u], L[pos] = L[pos], L[u]
            L[u], L[pos+1] = L[pos+1], L[u]
            pos += 1
        u += 1
```

2) On appellera plutôt `TriRapide` cette dernière fonction.

```
1  def TriRapide(p,q):
2      if q <= p:
3          return
4      pos = randint(p,q)
5      a = L[pos]
6      L[p], L[pos] = L[pos], L[p]
7      pos, u = p, p+1
8      while u <= q:
9          if L[u] < a and pos < q:
10             L[u], L[pos] = L[pos], L[u]
11             L[u], L[pos+1] = L[pos+1], L[u]
12             pos += 1
13         u += 1
14     TriRapide(p,pos-1)
15     TriRapide(pos+1,q)
```

Seules les lignes 2, 3 (condition d'arrêt) et 14, 15 (appels récursifs) ont été ajoutées.
Il faut retenir le principe du tri rapide et savoir le programmer. Pour appeler cette fonction sur une liste `L`, on lui fournira les paramètres d'entrée `0` et `len(L)-1`. Exemple :

```
1  L=[0,7,-3,2,9,11,5,8,4]
2  TriRapide(0,len(L)-1)
```

Simulations aléatoires élémentaires

Cours sur les simulations aléatoires élémentaires

I - Simulation aléatoire basique

Soit $p \in]0,1[$. Pour effectuer une instruction avec une probabilité p, on rappelle que l'on fait (en supposant la variable p préalablement affectée) :

```
1  from random import *
2
3  if random() < p:
4     # instruction(s) ici
```

Exemple : (Extrait oral Agro-Véto 2015)
Une urne contient initialement une boule blanche et une boule noire. On effectue une série de tirages aléatoires d'une boule jusqu'a obtenir une boule noire. A chaque tirage amenant une boule blanche, on

replace la boule blanche puis on multiplie par 2 le nombre de boules blanches présentes dans l'urne aprés la remise de la boule, puis on procède au tirage suivant.
Soit X la variable aléatoire égale au rang du tirage amenant une boule noire (si on obtient la boule noire), et qui vaut 0 si on n'obtient jamais de boule noire.

Compléter la fonction ci-dessous afin qu'elle réalise `m` fois l'expérience décrite ci-dessus (en arrêtant les tirages après l'obtention de `tmax` boules blanches), et renvoie la proportion des expériences où une boule noire a été obtenue :

```
def EstimeProbaEchec (m, tmax ):
    CompteSucces =0
    for i ...........:
        b =.........
        succes =0
        tirages =0
        while succes ==......... and tirages .........:
            tirages ...........
            if random ()..............:
                succes =1
            else :
                ..................
        CompteSucces += ..............
    return ...............................
```

Solution :
La variable `b` représente le nombre de boules blanches dans l'urne, `tirages` représente pour chacune des `m` expériences le nombre de tirages effectués, et, à chaque expérience, `succes` vaut 1 si et seulement si la boule noire a été tirée. La probabilité de tirer une boule noire est `1/(b+1)` car il y a dans l'urne exactement une boule noire et `b` boules blanches.

```python
def EstimeProbaEchec (m, tmax ):
    CompteSucces =0
    for i in range(m):
        b = 1
        succes = 0
        tirages = 0
        while succes == 0 and tirages < tmax:
            tirages += 1
            if random () < 1/(b+1):
                succes = 1
            else :
                b *= 2
        CompteSucces += succes
    return CompteSucces/m
```

On rappelle également que la fonction `randint(a,b)` appelée avec deux entiers a, b tels que $a \leqslant b$ fournit un entier choisi aléatoirement dans l'intervalle d'entiers $[\![a, b]\!]$. Cette fonction `randint` doit être importée de la bibliothèque `random`.

Exercice 1 :
Une urne contient initialement une boule blanche et une boule noire. On effectue dans cette urne une suite de tirages. À chaque tirage, on note la couleur de la boule tirée, on la remet dans l'urne et on ajoute en plus une boule noire.
On note X le rang de la première boule blanche tirée et Y le rang de la première boule noire tirée.
On note Z la variable aléatoire égale à $\min(\text{rgmax}, X)$ (plus petit des nombres rgmax et X) où rgmax est un entier naturel fixé (valeur seuil).
Écrire une fonction Python prenant rgmax en entrée, simulant cette expérience aléatoire, et donnant en sortie le couple (Z, Y).

Exercice 2 : (Extrait oral Agro-Véto 2016)
Soit N un entier naturel supérieur ou égal à 2. On considère N urnes, numérotées de 1 jusqu'à N, sachant que pour chaque i, l'urne numérotée i contient i jetons numérotés de 1 à i. On considère l'épreuve aléatoire consistant en une suite de tirages selon les règles suivantes :

- le premier tirage est effectué dans l'urne numérotée N ;
- si le jeton obtenu au k-ième tirage porte le numéro i, alors le (k + 1)-ème tirage est effectué dans l'urne numérotée i ;
- les différents jetons d'une même urne sont tirés équiprobablement.

1) Écrire une fonction Python prenant en argument un entier N, qui simule l'expérience ci-dessus, et renvoie le nombre de tirages nécessaires à l'obtention du premier 1.

2) On note Y_N le rang du tirage pour lequel on obtient le jeton 1 pour la première fois. On peut démontrer et nous l'admettons que :

$$E(Y_N) = 1 + \sum_{k=1}^{N-1} \frac{1}{k}.$$

Réaliser une simulation qui confirme graphiquement cette expression de $E(Y_N)$ en fonction de N.

Corrigés d'exercices sur les simulations aléatoires élémentaires

Exercice 1 :

Une urne contient initialement une boule blanche et une boule noire. On effectue dans cette urne une suite de tirages. À chaque tirage, on note la couleur de la boule tirée, on la remet dans l'urne et on ajoute en plus une boule noire.
On note X le rang de la première boule blanche tirée et Y le rang de la première boule noire tirée.
On note Z la variable aléatoire égale à $\min(\text{rgmax}, X)$ (plus petit des nombres rgmax et X) où rgmax est un entier naturel fixé (valeur seuil).
Écrire une fonction Python prenant rgmax en entrée, simulant cette expérience aléatoire, et donnant en sortie le couple (Z, Y).

Corrigé :

Tant que l'on n'a pas tiré au moins une boule blanche et une boule noire, on envisage à chaque tirage deux cas possibles (non disjoints) :

- soit on tire une boule blanche ou le nombre de tirage a dépassé la valeur limite rgmax,
- soit on tire une boule noire.

```python
# Exercice 1
from random import *

def Simule(rgmax):
    nbBlanches = nbNoires = 1
    blancheTiree = noireTiree = False
    numTirage = 0
    while not(blancheTiree and noireTiree):
        numTirage += 1
        p = random()
        print(p)
        # si on tire une boule blanche ou si on dépasse le seuil
        if p < nbBlanches/(nbBlanches + nbNoires) or numTirage >= rgmax:
            if not(blancheTiree):
                blancheTiree = True
                Z = numTirage
        # si on tire une boule noire
        if p >= nbBlanches/(nbBlanches + nbNoires):
            if not(noireTiree):
                noireTiree = True
                Y = numTirage
        nbNoires += 1
    return (Z,Y)
```

Exercice 2 : (Extrait oral Agro-Véto 2016)
Soit N un entier naturel supérieur ou égal à 2. On considère N urnes, numérotées de 1 jusqu'à N, sachant que pour chaque i, l'urne numérotée i contient i jetons numérotés de 1 à i. On considère l'épreuve aléatoire consistant en une suite de tirages selon les règles suivantes :

- le premier tirage est effectué dans l'urne numérotée N ;
- si le jeton obtenu au k-ième tirage porte le numéro i, alors le (k + 1)-ème tirage est effectué dans l'urne numérotée i ;
- les différents jetons d'une même urne sont tirés équiprobablement.

1) Écrire une fonction Python prenant en argument un entier N, qui simule l'expérience ci-dessus, et renvoie le nombre de tirages nécessaires à l'obtention du premier 1.
2) On note Y_N le rang du tirage pour lequel on obtient le jeton 1 pour la première fois. On peut démontrer et nous l'admettons que :

$$E(Y_N) = 1 + \sum_{k=1}^{N-1} \frac{1}{k}.$$

Réaliser une simulation qui confirme graphiquement cette expression de $E(Y_N)$ en fonction de N.

Corrigé :
1) Facile

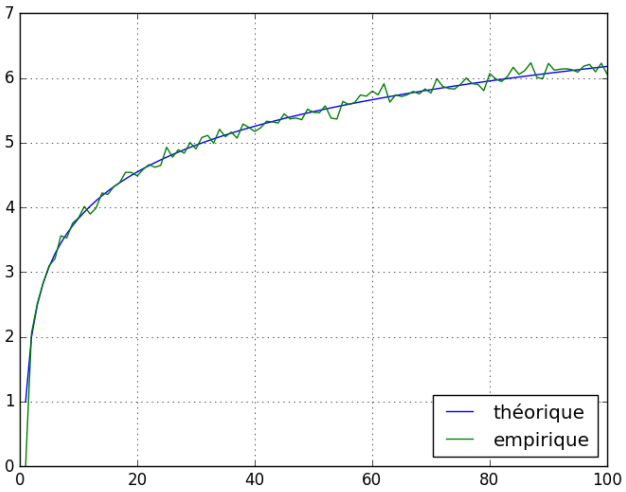
```python
# Question 1
def nbTirages(N):
    numTirage = 0
    numUrne = N
    while numUrne != 1:
        numUrne = randint(1,numUrne)
        numTirage += 1
    return numTirage
```

2) On crée deux courbes :

- la première donne les espérances théoriques $E(Y_k)$ de Y_k pour $k \in [\![1,\text{Nmax}]\!]$ en fonction de k.
- la seconde donne les moyennes empiriques $\frac{y_{k,1}+y_{k,2}+\cdots+y_{k,\text{nbSimulations}}}{\text{nbSimulations}}$ pour $k \in [\![1,\text{Nmax}]\!]$ où $y_{k,i}$ est le rang (simulé) du tirage pour lequel on obtient le jeton 1 pour la première fois dans le cas où $N = k$.
  Chaque valeur $y_{k,i}$ est obtenue par un appel `nbTirages(k)` à la fonction de la question précédente.

```python
# Question 2
from matplotlib import pyplot as plt

def esperance(Nmax, nbSimulations):
    abscisses = list(range(1,Nmax+1))
    ordonnees1 = [1]
    S = 1
    for k in range(1,Nmax):
        S += 1/k
        ordonnees1.append(S)
    ordonnees2 = []
    for k in range(1, Nmax + 1):
        S = 0
        for i in range(nbSimulations):
            S += nbTirages(k)
        ordonnees2.append(S/nbSimulations)
    plt.axis(xlim=(0.95,Nmax+0.05))
    plt.grid(True)
    plt.plot(abscisses, ordonnees1, label=r'théorique')
    plt.plot(abscisses, ordonnees2, label=r'empirique')
    plt.legend(loc='lower right')
    plt.show()
```

Voici un exemple de ce que l'on peut obtenir avec l'appel `esperance(100,1000)` :



Simulations aléatoires : lois discrètes

Cours sur les simulations de lois usuelles de BCPST1

Loi uniforme

Pour choisir un entier au hasard dans l'intervalle d'entier $[\![a, b]\!]$:

```
from random import randint
randint(a,b)
```

Plus généralement, pour choisir un élément quelconque d'un objet itérable (liste, tuple, chaîne de caractères...) :

```
from random import choice
objet1 = [5,8,11]
choice(objet1)
objet2 = "abcdkk"
choice(objet2)
```

Exercice 1 : Déplacement d'une puce dans le plan

Une puce se déplace dans le plan et se trouve à l'instant 0 au point de coordonnées $(0,0)$.
De l'instant t à $t+1$, la puce peut se déplacer sur l'un des 8 points voisins à coordonnées entières (à l'est, au nord-est, au nord, au nord-ouest, à l'ouest, au sud-ouest, au sud ou au sud-est).
Par exemple, si la puce est à la position (i,j) à l'instant t, elle se trouvera à l'instant $t+1$ à la position $(i,j+1)$ si elle se déplace au nord et à la position $(i-1,j-1)$ si elle se déplace au sud-ouest.
1) Sachant qu'à chaque instant t la puce se déplace d'un point à l'un des 8 points voisins de manière équiprobable, tracer un graphique représentant les mouvements de la puce simulés (pendant une durée T fournie en entrée de votre fonction Python. T doit être un entier naturel).
2) Écrire une fonction Python simulant les déplacements de la puce pendant une durée T et donnant en sortie le premier instant t où la puce se déplace sur un point déjà visité.

Loi de Bernoulli

Simulation d'une loi de Bernoulli

$p \in]0,1[$. On veut simuler X qui vaut 1 avec une probabilité p et 0 sinon.

```
from random import *
def Bernoulli(p):
    if random() < p:
        return 1
    return 0
```

Exercice 2 : Croissance d'un arbre

Soit $p \in]0,1[$. Un arbre est, à l'instant 0, réduit à un bourgeon au point $(0,0)$.
De l'instant t à $t+1$ chaque bourgeon (extrémité d'une branche) donne naissance à une nouvelle branche avec une probabilité p et à deux sinon. Chaque branche est dirigée en haut à gauche ou en haut à droite (les deux directions sont opposées si le bourgeon donne naissance à deux branches).
Si un bourgeon se trouve au point de coordonnées (i,j), l'extrémité d'une branche partant de ce bourgeon dirigée en haut à gauche aura pour coordonnées $(i-1,j+1)$ et l'extrémité d'une branche partant de ce bourgeon dirigée en haut à droite aura pour coordonnées $(i+1,j+1)$.
Écrire une fonction Python prenant T en entrée (un entier naturel) et p et représentant l'état de l'arbre après T unités de temps.
Conseil : on aura intérêt à tracer chaque branche au moment où elle est créée (donc autant de `plt.plot(...)` que de branches).

Loi binomiale

Simulation d'une loi binomiale

$p \in]0,1[$ et $n \in \mathbb{N}^*$. On veut simuler X qui vaut le nombre de succès au cours de n épreuves de Bernoulli indépendantes dont la probabilité de succès (pour chacune de ces épreuves) est p.

```
from random import *
def Binomiale(n,p):
    NbSucces = 0
    for k in range(n):
        if random() < p:
            NbSucces += 1
    return NbSucces
```

Exercice 3 : QCM

On pose 20 questions à un candidat. Pour chaque question, k réponses sont proposées, dont une seule est la bonne. Le candidat choisit au hasard une des réponses proposées.
On lui attribue un point par bonne réponse. Soit X_1 le nombre de points obtenus.
1) Écrire une fonction Python prenant k en entrée et simulant une réalisation de X_1.
2) Lorsque le candidat donne une mauvaise réponse, il peut choisir à nouveau une des autres réponses proposées. On lui attribue alors $\frac{1}{2}$ point par bonne réponse.
On note X le nombre de points total obtenus en tenant compte de cette nouvelle règle.
a) Écrire une fonction Python prenant k en entrée et simulant une réalisation de X.
b) Calculer empiriquement $E(X)$ pour $k \in [\![2,10]\!]$ (stocker le résultat sous forme de liste).
c) Tracer sur un même graphique les moyennes empiriques précédentes et $\frac{30}{k}$ pour $k \in [\![2,10]\!]$. Interpréter.

Loi hypergéométrique

Simulation d'une loi hypergéométrique

Une urne contient N boules dont a blanches et b noires. On note $p = \frac{a}{N}$ la proportion de boules blanches et on effectue n tirages successifs sans remise d'une boule de cette urne. On veut simuler le nombre X de boules blanches obtenues.

```python
# Loi hypergéométrique de paramètres N,n,p:
from random import *

def Hypergeo(N,n,p):
    a = N*p
    X = 0
    for k in range(n):
        if random() < a/N:
            X += 1
            a -= 1
        N -= 1
    return X
```

Exercice 4 :

Soit $n \in [\![1, 10]\!]$ fixé. Une urne contient 10 boules distinctes dont 3 boules blanches et 7 boules noires.
Une personne participe à un jeu qui se déroule en 4 parties successives.
Chaque partie consiste à extraire simultanément n boules de l'urne.
Les n boules sont replacées dans l'urne à l'issue de chacune des 4 parties.
Pour chacune de ces parties, le joueur gagne 1€ s'il obtient au moins deux boules blanches, et perd 2€ sinon.
1) Soit X la variable aléatoire égale au nombre de boules blanches extraites au cours de l'une des parties.
Écrire une fonction Python prenant n en entrée et simulant une réalisation de X.
2) Soit G le gain algébrique du joueur à l'issue du jeu (c'est-à-dire des quatre parties).
a) Écrire une fonction Python prenant n en entrée et simulant une réalisation de G.
b) Utiliser cette fonction pour en créer une autre calculant empiriquement $E(G)$.

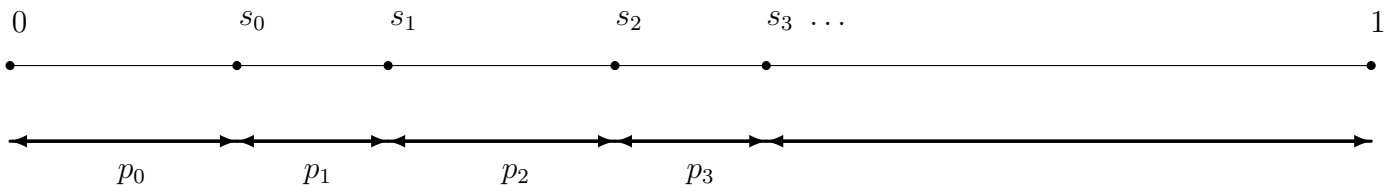
Cours sur les simulations de lois discrètes quelconques

Simulation d'une loi géométrique **(d'univers image $\mathbb{N}^*$)**

```python
from random import *

def Geom(p):
    rang = 1
    while random() > p: # tant qu'on a un "échec", on continue
        rang = rang + 1
    return rang
```

Exercice 1 : *Simulation d'une loi de Pascal.*
Soit $r \in \mathbb{N}^*$.
On lance une pièce de monnaie (truquée) dont la probabilité d'obtenir pile est p où $p \in]0,1[$.
On note X le nombre de lancers nécessaires pour obtenir le r-ième pile.
Écrire une fonction Python prenant r et p en entrée et simulant la variable aléatoire X.

Cas général : simulation d'une variable aléatoire discrète quelconque (introduction inspirée de http://bcpst.parc.free.fr/joomla/DOCUMENTS/Maths952/Info/TD6.pdf)
Soit X une variable aléatoire discrète. On note $X(\Omega) = \{x_n;\ n \in \mathbb{N}\}$ (éléments numérotés dans l'ordre croissant).
Il est aussi possible que $X(\Omega)$ soit fini, dans ce cas $X(\Omega) = \{x_0, x_1, \dots, x_m\}$.
Pour tout entier naturel n (ou tout entier de $[\![0, m]\!]$ dans le cas fini), on pose $p_n = P(X = x_n)$ et $s_n = \sum_{k=0}^{n} p_k$.
On découpe l'intervalle $[0,1]$ en un certain nombre d'intervalles de la façon suivante :



Pour simuler la variable X, on commence par choisir aléatoirement un réel u dans $]0,1[$.
Si $n \geqslant 1$, le réel u appartient à l'intervalle $[s_{n-1}, s_n[$ avec une probabilité $P(X = x_n) = p_n = s_n - s_{n-1}$, et il appartient à $[0, s_0[$ avec une probabilité $s_0 = p_0 = P(X = x_0)$. Ainsi, on simule X de la façon suivante :

- si $0 \leqslant u < p_0 = s_0$, on pose $X = x_0$
- sinon, on cherche $n \in \mathbb{N}^*$ tel que $s_{n-1} \leqslant u < s_n$ et on pose $X = x_n$.

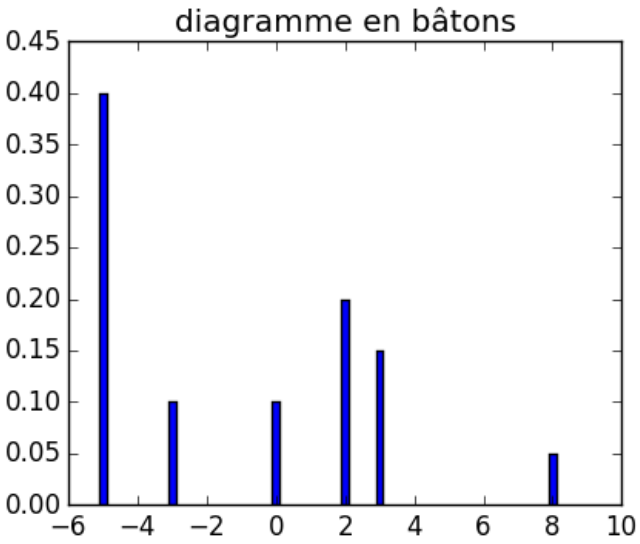
Exemple 1 :
Simulation de la variable aléatoire dont la loi de probabilité est donnée par :

x	-5	-3	0	2	3	8
$P(X = x)$	$0,4$	$0,1$	$0,1$	$0,2$	$0,15$	$0,05$

Principe :

1. On saisit les listes correspondant à l'univers image et aux probabilités $P(X = x)$ correspondantes. On écrit ensuite la fonction prenant en entrée ces deux listes, celle-ci est définie comme indiqué ci-dessous.
2. (a) On génère un réel u choisi aléatoirement dans $]0, 1[$.
   (b) On initialise i à 0 et on pose $s_i = s_0 = p_0$. Tant que $s_i \leqslant u$, on augmente i d'une unité et on met à jour s_i (égal à $\sum_{k=1}^{i} p_i = p_i + s_{i-1}$).
   (c) On donne en retour x_i où i est le dernier indice calculé précédemment (celui ayant fait sortir de la boucle).
3. On appelle la fonction avec les deux listes créées au début.

```
1  from random import *
2  from matplotlib import pyplot as plt
3
4  Valeurs = [-5,-3,0,2,3,8]
5  Probas = [0.4,0.1,0.1,0.2,0.15,0.05]
6
7  def Simul(Val, Prob):
8      S, i = Prob[0], 0
9      u = random()
10     while S <= u:
11         i += 1
12         S += Prob[i]
13     return Val[i]
14
15 # Appel à la fonction
16 print(Simul(Valeurs, Probas))
```



Exercice 2 :

On s'est intéressé à la couleur de cheveux d'un échantillon de 5904 sujets d'une population et on a obtenu le tableau ci-dessous :

Couleur des cheveux	Nombre de sujets présentant cette couleur
blond	2365
brun	2487
noir	954
roux	98
Total	5904

Soit X la variable aléatoire donnant la couleur de cheveux d'un sujet choisi au hasard dans cette population. En vous basant sur le tableau ci-dessus, proposer une simulation de cette variable aléatoire.

Exemple 2 : Simulation d'une loi de Poisson

On s'appuie sur le principe précédent : on calcule $s_n = \sum_{k=0}^{n} p_k = \sum_{k=0}^{n} e^{-\lambda}\frac{\lambda^k}{k!}$ jusqu'à ce que $s_n > u$ (avec ici $p_k = P(X = k)$), on renvoie alors n.

```
from random import *
from math import *
def Poisson(lambd):
    u = random()
    aAjouter = exp(-lambd)
    FreqCum = aAjouter
    n = 0
    while FreqCum <= u:
        n += 1
        aAjouter *= lambd/n
        FreqCum += aAjouter
    return n
```

(autre façon de faire pour la loi de Poisson : simuler une loi binomiale $\mathcal{B}\left(n, \frac{\lambda}{n}\right)$ avec n grand car alors $\mathcal{B}\left(n, \frac{\lambda}{n}\right) \simeq \mathcal{P}(\lambda)$)

Tracé d'un diagramme en barres :

Pour tracer des barres de hauteurs $y_0, y_1, \ldots, y_n$ (où $y_i \geqslant 0$ pour tout $i \in [\![0, n]\!]$) aux abscisses $x_0, x_1, \ldots, x_n$, on utilisera l'instruction :

```
plt.bar(abscisses, hauteurs, width=0.2, color="green")
```

où `hauteurs` est la liste des hauteurs et `abscisses` est la liste des abscisses. L'argument optionnel `color` définit la couleur des barres (ici elles sont vertes) et `width` la largeur des barres.
Par exemple, taper dans l'éditeur le code ci-dessous et voyez ce qu'il se passe :

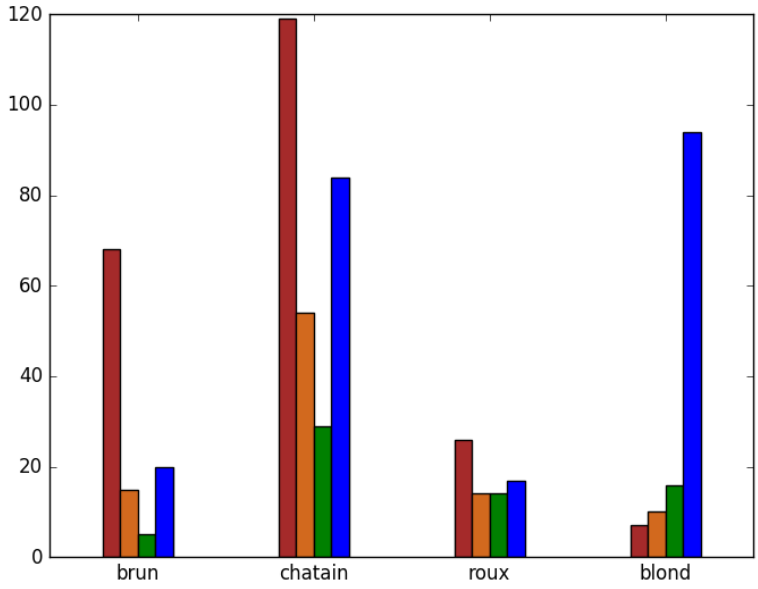
```
1  from matplotlib import pyplot as plt
2
3  # ajuster la fenêtre graphique
4  plt.axis(xmin=2, xmax=10, ymin=0, ymax = 10)
5
6  # Données
7  abscisses = [3,6,9]
8  hauteurs = [3,5,8]
9  largeur = 0.4  # largeur des barres
10
11 # tracer le diagramme avec des noms pour les barres
12 plt.bar(abscisses, hauteurs, width=largeur, label="Trois barres", align="center")
13 plt.xticks(abscisses, ("Barre 1", "Barre 2", "Barre 3"))
14 plt.legend(loc="upper right")
15
16 plt.show()
```

Par défaut, les barres sont alignées sur leur côté gauche. Voyez par exemple ce qu'il se passe en supprimant la portion de code `, align="center"` de la ligne 12.

Exercice 3 :
Le tableau de contingence ci-dessous obtenu sur un échantillon de 592 femmes suivant la couleur de leurs yeux et la couleur de leurs cheveux.

Yeux \ Cheveux	brun	chatain	roux	blond
marron	68	119	26	7
noisette	15	54	14	10
vert	5	29	14	16
bleu	20	84	17	94

Tracer un diagramme en barres représentant ces données (barres de largeurs 0.1). On veut un résultat de ce type en sortie (chaque groupe de barres correspond à un groupe de couleurs d'yeux pour une couleur de cheveux donnée).



Pour les couleurs des barres (marron, noisette, vert, bleu), on utilisera respectivement les noms suivants : brown, chocolate, green, blue.

Exercice 4 :
1) Écrire une fonction Python prenant $\lambda > 0$ et $n \in \mathbb{N}^*$ en entrée, simulant n réalisation d'une variable aléatoire X suivant la loi de Poisson $\mathcal{P}(\lambda)$ et donnant en sortie la liste `L` de 51 éléments telle que `L[k]` corresponde à la fréquence de la valeur k ($0 \leqslant k \leqslant 50$) (nombre de fois où la valeur k a été prise par X, divisé par n).
2) Écrire une fonction Python prenant $\lambda > 0$ en entrée et traçant sur un même graphique le diagramme en barres d'une loi de Poisson $\mathcal{P}(\lambda)$ théorique (la barre centrée en k aura pour hauteur $e^{-\lambda}\frac{\lambda^k}{k!}$) et celui de la même loi simulée avec un échantillon de 10000 réalisations (s'inspirer de la forme du diagramme de l'exercice précédent).

Corrigés d'exercices sur les simulations de lois discrètes

Exercice 1 : Croissance d'un arbre
Soit $p \in]0,1[$. Un arbre est, à l'instant 0, réduit à un bourgeon au point $(0,0)$.
De l'instant t à $t+1$ chaque bourgeon (extrémité d'une branche) donne naissance à une nouvelle branche avec une probabilité p et à deux sinon. Chaque branche est dirigée en haut à gauche ou en haut à droite (les deux directions sont opposées si le bourgeon donne naissance à deux branches).
Si un bourgeon se trouve au point de coordonnées (i,j), l'extrémité d'une branche partant de ce bourgeon dirigée en haut à gauche aura pour coordonnées $(i-1, j+1)$ et l'extrémité d'une branche partant de ce bourgeon dirigée en haut à droite aura pour coordonnées $(i+1, j+1)$.
Écrire une fonction Python prenant T en entrée (un entier naturel) et p et représentant l'état de l'arbre après T unités de temps.
Conseil : on aura intérêt à tracer chaque branche au moment où elle est créée (donc autant de `plt.plot(...)` que de branches).

Corrigé :
On stocke dans une liste `Bourgeons` les coordonnées des bourgeons à l'instant t et dans une autre liste `BourgeonsBis` les coordonnées des prochains bourgeons (à l'instant $t+1$).

```
# Exercice 1
from random import *
from matplotlib import pyplot as plt

def Arbre(T,p):
    P1 = (0,0)
    Bourgeons = [P1]
    for t in range(T):
        BourgeonsBis = []
        for b in Bourgeons:
            if random() < p:
                if random() < 1/2:
                    bvoisin = (b[0]-1,b[1]+1)
                else:
                    bvoisin = (b[0]+1,b[1]+1)
                BourgeonsBis.append(bvoisin)
                plt.plot([b[0],bvoisin[0]],[b[1],bvoisin[1]])
            else:
                bvoisin1 = (b[0]-1,b[1]+1)
                BourgeonsBis.append(bvoisin1)
                plt.plot([b[0],bvoisin1[0]],[b[1],bvoisin1[1]])
                bvoisin2 = (b[0]+1,b[1]+1)
                BourgeonsBis.append(bvoisin2)
                plt.plot([b[0],bvoisin2[0]],[b[1],bvoisin2[1]])
        Bourgeons = BourgeonsBis
    plt.show()
```

Exercice 2 : QCM

On pose 20 questions à un candidat. Pour chaque question, k réponses sont proposées, dont une seule est la bonne. Le candidat choisit au hasard une des réponses proposées. On lui attribue un point par bonne réponse. Soit X_1 le nombre de points obtenus.

1) Écrire une fonction Python prenant k en entrée et simulant une réalisation de X_1.

2) Lorsque le candidat donne une mauvaise réponse, il peut choisir à nouveau une des autres réponses proposées. On lui attribue alors $\frac{1}{2}$ point par bonne réponse. On note X le nombre de points total obtenus en tenant compte de cette nouvelle règle.

a) Écrire une fonction Python prenant k en entrée et simulant une réalisation de X.

b) Calculer empiriquement $E(X)$ pour $k \in [\![2, 10]\!]$ (stocker le résultat sous forme de liste).

c) Tracer sur un même graphique les moyennes empiriques précédentes et $\frac{30}{k}$ pour $k \in [\![2, 10]\!]$.

Interpréter.

Corrigé :

```
# Exercice 2

from random import *

# Question 1

def X1(k):
    x1 = 0
    for j in range(20):
        if random() < 1/k:
            x1 += 1
    return x1

# Question 2 a

def X(k):
    x = 0
    for j in range(20):
        if random() < 1/k:
            x += 1
        elif random() < 1/(k-1):
            x += 0.5
    return x
```

Pour les questions 2) b) et 2) c), nos fonctions prennent en variable d'entrée `nbSimul` qui est un entier naturel déterminant combien de fois on simule la variable aléatoire (pour calculer la moyenne des valeurs de ces simulations).

```python
# Question 2 b

def EX(nbSimul):
    L = []
    for k in range(2,11):
        s = 0
        for j in range(nbSimul):
            s += X(k)
        L.append(s/nbSimul)
    return L

# Question 2 c

def Graphes(nbSimul):
    X = list(range(2,11))
    Y1 = EX(nbSimul)
    Y2 = [30/k for k in X]
    plt.grid = True
    plt.axis(xmin = 1, xmax = 11)
    plt.plot(X,Y1,label="Moyennes empiriques")
    plt.plot(X,Y2, label="y = 30/k")
    plt.legend(loc="upper right")
    plt.show() # On conjecture que 30/k est l'espérance théorique de X.
```

Exercice 3 :

On s'est intéressé à la couleur de cheveux d'un échantillon de 5904 sujets d'une population et on a obtenu le tableau ci-dessous :

Couleur des cheveux	Nombre de sujets présentant cette couleur
blond	2365
brun	2487
noir	954
roux	98
Total	5904

Soit X la variable aléatoire donnant la couleur de cheveux d'un sujet choisi au hasard dans cette population. En vous basant sur le tableau ci-dessus, proposer une simulation de cette variable aléatoire.

Corrigé :

On doit simuler la variable aléatoire qualitative de loi approximative donnée ci-dessous :

x	blond	brun	noir	roux
$P(X=x)$	$\frac{2365}{5904}$	$\frac{2487}{5904}$	$\frac{954}{5904}$	$\frac{98}{5904}$

On sait comment faire cela. Il nous suffit d'adapter le code connu qu'on applique avec la liste des modalités (les couleurs des cheveux) et la liste des probabilités correspondantes.

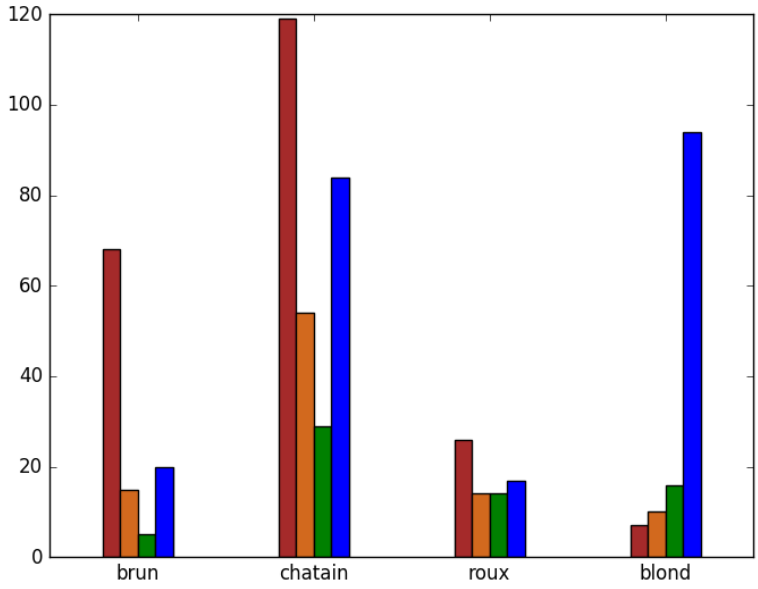
```
# Exercice 3
from random import *
from matplotlib import pyplot as plt

Valeurs = ["blond","brun","noir","roux"]
Effectifs = [2365,2487,954,98]
Total = 5904
Probas = [Eff/Total for Eff in Effectifs]

def Simul(Val, Prob):
    S, i = Prob[0], 0
    u = random()
    while S <= u:
        i += 1
        S += Prob[i]
    return Valeurs[i]

# Appel à la fonction
print(Simul(Valeurs, Probas))
```

Exercice 4 :

Le tableau de contingence ci-dessous obtenu sur un échantillon de 592 femmes suivant la couleur de leurs yeux et la couleur de leurs cheveux.

Yeux \ Cheveux	brun	chatain	roux	blond
marron	68	119	26	7
noisette	15	54	14	10
vert	5	29	14	16
bleu	20	84	17	94

Tracer un diagramme en barres représentant ces données (barres de largeurs 0.1). On veut un résultat de ce type en sortie (chaque groupe de barres correspond à un groupe de couleurs d'yeux pour une couleur de cheveux donnée).



Pour les couleurs des barres (marron, noisette, vert, bleu), on utilisera respectivement les noms suivants : brown, chocolate, green, blue.

Corrigé :
On trace 4 diagrammes en barres : les effectifs par couleurs de cheveux pour chacune des 4 couleurs d'yeux données.
Les barres devant être de largeur 0.1, on aligne les sommets inférieurs gauches des 4 barres du k-ième groupe aux points d'abscisses $k-0.2$, $k-0.1$, k et $k+0.1$ (de sorte que ce groupe soit centré sur le point d'abscisse k).

```
# Exercice 4
import numpy as np
largeur = 0.1
abs = np.array([1,2,3,4])
CouleursCheveux = ("brun","chatain","roux","blond")
CouleursYeux = ("brown","chocolate","green","blue")
Hauteurs = ([68,119,26,7],[15,54,14,10],[5,29,14,16],[20,84,17,94])

for k in range(4):
    plt.bar(abs-0.2+0.1*k, Hauteurs[k], width=largeur, color=CouleursYeux[k])

plt.xticks(abs, CouleursCheveux) # légende sous les groupes de barres.

plt.show()
```

Représentation des graphes

Cours sur les représentations des graphes

Introduction sur les graphes

Un graphe peut être décrit comme un ensemble de noeuds et d'arêtes orientées (ou non) munies éventuellement de poids.

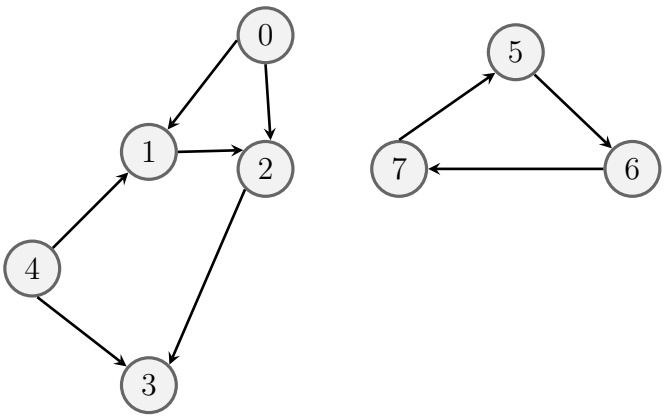


Figure 1 : Un graphe orienté

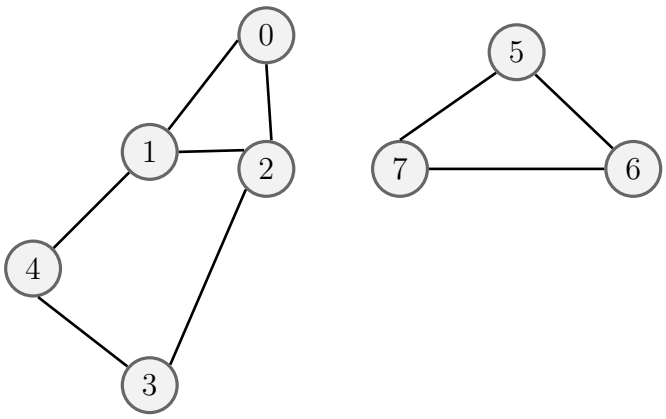


Figure 2 : Un graphe non orienté

Un graphe (orienté) pondéré peut servir par exemple à représenter un réseau routier où les poids représenteraient des distances entre villes (les villes seraient les noeuds).

Exemple

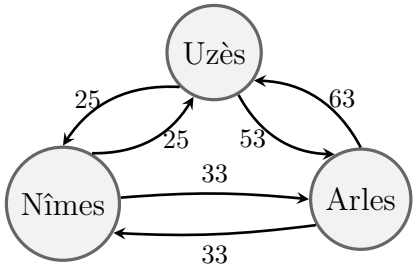


Figure 3 : Un graphe orienté pondéré

Un graphe non orienté peut être vu comme un graphe orienté où chaque arête reliant un noeud i à j dans la version non orienté est remplacée par un couple d'arête reliant i à j et j à i dans la version orientée du graphe.

Exemple

Les graphes interviennent dans de nombreux algorithmes répondant à des problématiques concrètes. A ce titre, ils peuvent s'avérer utiles, voire incontournables dans l'élaboration des projets.
On peut représenter un graphe non pondéré possédant N noeuds au moins de trois façons :

❶ Par la donnée de son nombre N de noeuds (numérotés de 0 à $N-1$), et de ses arêtes sous la forme de couples (i,j) indiquant qu'il existe une arête reliant le noeud i au noeud j (avec $i,j \in [\![0,N-1]\!]$).

❷ Par la donnée d'une matrice carrée M de taille N telle que, pour $i,j \in [\![0,N-1]\!]$,
$\begin{cases} M[i,j]=1 \text{ s'il existe une arête reliant le noeud } i \text{ au noeud } j, \\ M[i,j]=0 \text{ sinon.} \end{cases}$
M est alors appelée la *matrice d'adjacence du graphe.*

❸ Par la donnée d'une liste L de listes telle que, pour $i \in [\![0,N-1]\!]$, la liste $L[i]$ est constituée des noeuds extrémités d'arêtes partant du noeud i.
Par exemple, si $L[3]=[1,4]$, cela signifie que les arêtes partant du noeud 3 sont les arêtes $(3,1)$ et $(3,4)$. La liste L est appelée la *liste d'adjacence du graphe.*

Exemple :
Le graphe de la figure 1 peut être représenté par :

❶ :

N=8	(0,1)	(0,2)	(1,2)
	(2,3)	(4,1)	(4,3)
	(5,6)	(6,7)	(7,5)

, ❷ : $\begin{pmatrix} 0&1&1&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&1&0&1&0&0&0&0 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&1&0&0 \end{pmatrix}$ et ❸ :

```
L=[[1,2],
   [2],
   [3],
   [],
   [1,3],
   [6],
   [7],
   [5]]
```

.

Le graphe de la figure 2 peut être représenté par :

❶ :

N=8	(0,1)	(0,2)	(1,2)
	(2,3)	(4,1)	(4,3)
	(5,6)	(6,7)	(7,5)

, ❷ : $\begin{pmatrix} 0&1&1&0&0&0&0&0 \\ 1&0&1&0&1&0&0&0 \\ 1&1&0&1&0&0&0&0 \\ 0&0&1&0&1&0&0&0 \\ 0&1&0&1&0&0&0&0 \\ 0&0&0&0&0&0&1&1 \\ 0&0&0&0&0&1&0&1 \\ 0&0&0&0&0&1&1&0 \end{pmatrix}$ et ❸ :

```
L=[[1,2],
   [0,2,4],
   [0,1,3],
   [2,4],
   [1,3],
   [6,7],
   [5,7],
   [5,6]]
```

.

Pour les graphes pondérés, on complète les représentations ❶, ❷ et ❸ ainsi :

- On remplace les couples (i,j) de ❶ par des triplets (i,j,p) signifiant l'existence d'une arête de poids p reliant le noeud i au noeud j.
- La matrice d'adjacence M est telle que :
$\begin{cases} M[i,j]=p \text{ s'il existe une arête de poids } p \text{ reliant le noeud } i \text{ au noeud } j, \\ M[i,j]=0 \text{ sinon.} \end{cases}$

- Dans la liste L d'adjacence, pour $i \in [\![0, N-1]\!]$, la liste $L[i]$ est formée des couples (j, p) signifiant qu'il existe une arête de poids p reliant le noeud i au noeud j.

Exemple :
En choisissant les numéros $0, 1, 2$ respectivement pour Uzès, Nîmes et Arles, le graphe de la figure 3 peut être représenté par :

❶ : `N=3 (0,1,25) (1,0,25) (0,2,53) (2,0,63) (1,2,33) (2,1,33)`, ❷ : $\begin{pmatrix} 0 & 25 & 53 \\ 25 & 0 & 33 \\ 63 & 33 & 0 \end{pmatrix}$ et ❸ :

```
L=[[(1,25),(2,53)],
   [(0,25),(2,33)],
   [(0,63),(1,33)]]
```

.

Listons enfin les avantages et inconvénients des trois représentations (en signalant tout de même la supériorité de la représentation ❸ dans la majorité des cas) :

	Représentation ❶	Représentation ❷ par matrice d'adjacence.	Représentation ❸ par liste d'adjacence.
Avantages	Représentation facile à obtenir à partir d'un graphe : il suffit de lister ses arêtes. Représentation pratique pour dessiner un graphe : on dessine les arêtes une à une.	Représentation facile à exploiter dans un programme informatique.	Représentation pratique à exploiter dans le cadre d'un programme informatique. Les noeuds accessibles par une arête à partir d'un noeud donné sont rapidement retrouvés.
Inconvénients	Représentation non adaptée à l'exploitation du graphe dans un programme informatique.	Assez lent à exploiter dans le cadre d'un programme informatique si le graphe possède beaucoup de noeuds : pour déterminer l'ensemble des noeuds accessibles par une arête partant d'un noeud donné, on doit considérer tous les noeuds du graphe.	Représentation un peu moins facile à exploiter qu'une matrice d'adjacence.

Terminons par la mise en pratique de ces diverses représentations (il faut savoir "jongler" entre elles). C'est aussi l'occasion de manipuler de nouveau des listes et des tableaux.

Exercice 1 :
On suppose donné un graphe non pondéré G par la représentation ❶ sous forme d'une liste $L = [N, (i_1, j_1), (i_2, j_2), \dots, (i_p, j_p)]$ où N est le nombre de noeuds et les (i_k, j_k) $(k \in [\![1, p]\!])$ sont les p arêtes.

1. Ecrire une fonction Python prenant en entrée L et donnant en sortie la matrice d'adjacence du graphe.
2. Ecrire une fonction Python prenant en entrée L et donnant en sortie la liste d'adjacence du graphe.

Exercice 2 :

1. Ecrire une fonction Python prenant en entrée la matrice d'adjacence d'un graphe non pondéré et donnant en sortie la liste d'adjacence du graphe.
2. Ecrire une fonction Python prenant en entrée la liste d'adjacence d'un graphe non pondéré et donnant en sortie la matrice d'adjacence du graphe.

Exercice 3 :
Adapter les réponses des exercices précédents au cas des graphes pondérés.

Corrigés d'exercices sur les représentations des graphes

Corrections des exercices :

On commence par importer numPy avec l'abréviation usuelle.

```
import numpy as np
```

Exercice 1 :
Question 1
L'idée est de parcourir les couples (i, j) dans la liste L fournie en entrée et de placer des 1 à la ligne i et colonne j correspondante de la matrice d'adjacence.
On rappelle pour cela qu'on peut itérer sur les éléments d'une liste L à l'aide de

```
for elem in L:
```

et que, si L est de taille n, et si i, j sont deux entiers appartenant à $[\![0, n-1]\!]$, alors L[i :], L[:j], L[i :j] sont des sous-listes de L formées respectivement des éléments de L à partir de celui d'indice i, jusqu'à celui d'indice $j-1$, et entre celui d'indice i et celui d'indice $j-1$.

```python
# Exercice 1, question 1

def MatAdj(L):
    nbNoeuds = L[0]
    MatriceAdj = np.zeros([nbNoeuds,nbNoeuds])
    for couple in L[1:]:
        MatriceAdj[couple] = 1
    return MatriceAdj
```

Question 2

Si L est une liste de couples (i, j), on peut récupérer les éléments i et j des couples en itérant sur L de cette manière

```python
for i,j in L:
```

On initialise la liste d'adjacence à une liste de n sous-listes vides où n est le nombre de noeuds du graphe. On met le noeud j dans la sous-liste i si on rencontre le couple (i, j) dans L.

```python
# Exercice 1, question 2

def LAdj(L):
    nbNoeuds = L[0]
    ListeAdj = [[] for k in range(nbNoeuds)]
    for noeudOrig, noeudExt in L[1:]:
        ListeAdj[noeudOrig].append(noeudExt)
    return ListeAdj
```

Exercice 2 :

Question 1

On rappelle simplement ici que, si A est une matrice, A.shape donne en retour la liste (nblig,nbcol) formée du nombre de lignes et de colonnes de la matrice. L'idée consiste à placer le noeud j dans la sous-liste i chaque fois qu'un coefficient $M[i, j]$ égal à 1 est rencontré dans la matrice d'adjacence M.

```python
# Exercice 2, question 1
def deMataListAdj(M):
    nbNoeuds = M.shape[0]
    ListeAdj = [[] for k in range(nbNoeuds)]
    for lig in range(nbNoeuds):
        for col in range(nbNoeuds):
            if M[lig,col] == 1:
                ListeAdj[lig].append(col)
    return ListeAdj
```

Question 2

On effectue le procédé réciproque. On parcourt chaque sous-liste de la liste d'adjacence. Si la i-ième sous-liste possède j, on place un 1 dans la matrice d'adjacence à la i-ème ligne et j-ème colonne.

```python
# Exercice 2, question 2
def deListAdjaMat(L):
    nbNoeuds = len(L)
    MatriceAdj = np.zeros([nbNoeuds,nbNoeuds])
    for noeud in range(nbNoeuds):
        for noeudExt in L[noeud]:
            MatriceAdj[noeud, noeudExt] = 1
    return MatriceAdj
```

Exercice 3 :

Il n'y a pas de difficultés particulières si l'on a bien compris les deux exercices précédents.

```python
# Exercice 3

# Exercice 1 avec poids
def MatAdjBis(L):
    nbNoeuds = L[0]
    MatriceAdj = np.zeros([nbNoeuds,nbNoeuds])
    for triplet in L[1:]:
        MatriceAdj[couple] = triplet[2]
    return MatriceAdj

def LAdjBis(L):
    nbNoeuds = L[0]
    ListeAdj = [[] for k in range(nbNoeuds)]
    for noeudOrig, noeudExt, poids in L[1:]:
        ListeAdj[noeudOrig].append((noeudExt,poids))
    return ListeAdj

# Exercice 2 avec poids

def deMataListAdjBis(M):
    nbNoeuds = M.shape[0]
    ListeAdj = [[] for k in range(nbNoeuds)]
    for lig in range(nbNoeuds):
        for col in range(nbNoeuds):
            if M[lig,col] != 0:
                ListeAdj[lig].append((col,M[lig,col]))
    return ListeAdj

def deListAdjaMatBis(L):
    nbNoeuds = len(L)
    MatriceAdj = np.zeros([nbNoeuds,nbNoeuds])
    for noeud in range(nbNoeuds):
        for noeudExt, poids in L[noeud]:
            MatriceAdj[noeud, noeudExt] = poids
    return MatriceAdj
```

Simulation de lois à densité

Cours sur les simulations de lois à densité

1) Tracé d'un histogramme

Point méthode : Tracer un histogramme

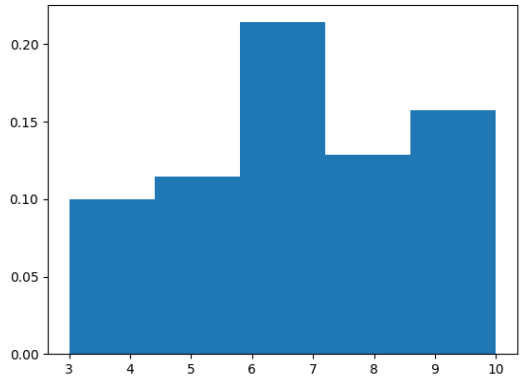
Pour tracer l'histogramme d'une série de valeurs d'une variable aléatoire continue, on utilisera la méthode `hist` de `matplotlib.pyplot`.
Pour que l'histogramme ait k barres et une aire totale égale à 1, on pourra écrire :

```
import matplotlib.pyplot as plt
plt.hist(Valeurs, bins=k, normed=1) # tracé de l'histogramme
```

Ici `Valeurs` est la liste des valeurs de la variable aléatoire.

Exemple :

```
from random import *
import matplotlib.pyplot as plt
Valeurs = [randint(3,10) for k in range(50)]
plt.hist(Valeurs, bins=5, normed=1)
plt.show()
```



2) Simulation d'une loi quelconque

a) Simulation d'une loi uniforme sur $[0,1]$.

Pour choisir un nombre au hasard dans $[0,1]$, on utilise la fonction random() de la bibliothèque `random`.

```
from random import *

def Unif():
    return random()
```

b) Cas général

Soit X une variable aléatoire de fonction de répartition $F : \mathbb{R} \to [0,1]$.
On suppose que F réalise une bijection strictement croissante d'un intervalle $]a,b[$ sur $]0,1[$.
Alors, en notant $F^{-1} :]0,1[\to]a,b[$ la bijection réciproque de F, et U une variable aléatoire de loi uniforme sur $]0,1[$,

$$F^{-1}(U) \text{ suit la même loi que } X.$$

Ainsi, si l'on note RecF la bijection réciproque F^{-1}, nous pouvons utiliser le code ci-dessous :

```
from random import *

def Simul(RecF):
    return RecF(random())
```

Ce résultat offre un procédé bien pratique pour simuler avec Python la réalisation
d'une telle variable aléatoire X si l'on sait expliciter F^{-1} : on choisit au hasard un réel u dans $]0,1[$ et on donne la valeur de $F^{-1}(u)$.

Preuve

Posons $Y = F^{-1}(U)$. L'univers image de Y est $]a,b[$. Soit $x \in]a,b[$.

$$P(Y \leqslant x) = P(F^{-1}(U) \leqslant x) = P(U \leqslant F(x)) = F(x) = P(X \leqslant x).$$

X et Y ont presque sûrement le même univers image et la même fonction de répartition. La loi de Y est donc
bien celle de X.

c) Applications

- **Simulation d'une loi uniforme sur** $[a,b]$.

  On peut commencer par remarquer que :

  $$x \in [0,1] \Leftrightarrow a + (b-a)x \in [a,b]$$

  donc choisir au hasard un nombre réel entre a et b revient à générer $a + (b-a)x$ où x a été choisi au hasard dans $[0,1]$. D'où le code ci-dessous :

```
from random import *

def Unif(a,b):
    u = random()
    return a + (b-a)*u
```

  On peut aussi retrouver ce résultat grâce au résultat théorique précédent.

  La fonction de répartition d'une variable aléatoire suivant une loi uniforme sur $[a,b]$ est F définie par :

  $$\begin{cases} F(x) = 0 \text{ si } x < a \\ F(x) = \frac{x-a}{b-a} \text{ si } x \in [a,b] \\ F(x) = 1 \text{ si } x > b \end{cases}.$$

Elle réalise une bijection strictement croissante de $]a,b[$ sur $]0,1[$.

Déterminons la bijection réciproque F^{-1}. Soit $x \in]a,b[$ et $y \in]0,1[$.

$$F(x) = y \Leftrightarrow \frac{x-a}{b-a} = y \Leftrightarrow x - a = y(b-a) \Leftrightarrow x = a + y(b-a).$$

On en déduit $\forall y \in]0,1[$, $F^{-1}(y) = a + y(b-a)$.

Ainsi, si U suit la loi uniforme sur $]0,1[$, alors $a + (b-a)U$ suit la loi uniforme sur $]a,b[$.

- **Simulation d'une loi exponentielle de paramètre λ.**

Application : Simulation d'une loi exponentielle.

Soit X une variable aléatoire de loi exponentielle de paramètre $\lambda > 0$.

On sait que sa fonction de répartition F est donnée par $F(x) = \begin{cases} 0 & \text{si } x < 0 \\ 1 - e^{-\lambda x} & \text{si } x \geqslant 0 \end{cases}$

et réalise une bijection strictement croissante de $]0,+\infty[$ sur $]0,1[$.

Déterminons la bijection réciproque F^{-1}. Soit $x \in]0,+\infty[$ et $y \in]0,1[$.

$$\begin{aligned} F(x) = y &\Leftrightarrow 1 - e^{-\lambda x} = y \Leftrightarrow e^{-\lambda x} = 1 - y \Leftrightarrow -\lambda x = \ln(1-y) \\ &\Leftrightarrow x = -\frac{\ln(1-y)}{\lambda}. \end{aligned}$$

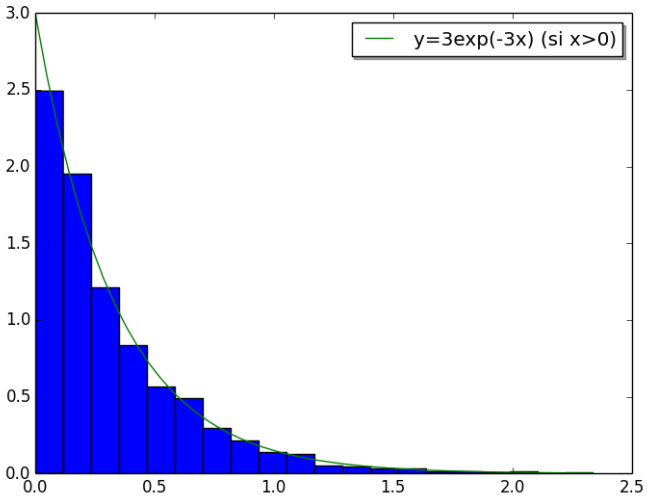
On en déduit $\forall y \in]0,1[$, $F^{-1}(y) = -\frac{\ln(1-y)}{\lambda}$. Ainsi, si U suit la loi uniforme sur $]0,1[$, alors $-\frac{\ln(1-U)}{\lambda}$ suit la loi exponentielle de paramètre $\lambda > 0$. Comme il est clair que $1-U$ et U suivent la même loi, on en déduit :

Si U suit la loi uniforme sur $]0,1[$, alors $-\frac{\ln(U)}{\lambda}$ suit la loi exponentielle de paramètre λ

La fonction ci-dessous permet donc de simuler une variable aléatoire de loi exponentielle de paramètre $\lambda > 0$.

```
1 from random import *
2 from math import *
3
4 def expo(lamb):
5     u = random()
6     return -log(u)/lamb
```

On peut l'appeler plusieurs fois et représenter sur un même graphique la répartition des valeurs obtenues (sous forme d'histogramme) et la densité de la loi exponentielle simulée.




```python
import numpy as np
import matplotlib.pyplot as plt

Valeurs = np.array([expo(3) for k in range(1000)]) # 1000 simulations de X
plt.hist(Valeurs, bins = 20, normed=1) # tracé de l'histogramme
X = np.linspace(0,np.amax(Valeurs))
Y = 3*np.exp(-3*X)  # ordonnées des points de la densité
plt.plot(X,Y,label = 'y=3exp(-3x) (si x>0)') # tracé de la densité
plt.legend(loc='upper right', shadow=True) # légende (en haut à droite)
plt.show()
```

d) Cas où on ne sait pas calculer la bijection réciproque.

Indiquons comment faire lorsque la fonction réciproque F^{-1} de la fonction de répartition est inconnue.
Soit X une variable aléatoire de fonction de répartition $F : \mathbb{R} \to [0,1]$. On suppose de plus que F réalise une bijection strictement croissante d'un intervalle $]a,b[$ sur $]0,1[$ et on note F^{-1} la bijection réciproque.
Si U suit la loi uniforme sur $]0,1[$, on sait que $F^{-1}(U)$ a même loi que X.
De plus, si $x \in]a,b[$ et $u \in]0,1[$,

$$x = F^{-1}(u) \Leftrightarrow F(x) = u \Leftrightarrow F(x) - u = 0$$

Ainsi, même si l'on ne sait pas expliciter F^{-1}, si u est un réel choisi au hasard dans $]0,1[$, on peut obtenir une approximation de $F^{-1}(u)$ comme unique solution x de $F(x) - u = 0$ par un algorithme de type dichotomie ou méthode de Newton.

Rappel : Algorithme de dichotomie

On considère une fonction f continue sur un segment $[a,b]$.

On suppose que f s'annule exactement une fois sur $[a,b]$, en un point que l'on note β.

On définit les suites $(a_k)_{k\geqslant 0}$ et $(b_k)_{k\geqslant 0}$ de la façon suivante :

- $a_0 = a$ et $b_0 = b$
- Pour tout entier naturel k, on note $c_k = \frac{a_k+b_k}{2}$ et :

si $f(a_k)f(c_k) \leqslant 0$, alors $a_{k+1} = a_k$ et $b_{k+1} = c_k$

sinon $a_{k+1} = c_k$ et $b_{k+1} = b_k$

On sait alors que les suites (a_k) et (b_k) convergent toutes les deux vers β, en vérifiant :

$\forall k \in \mathbb{N}$, $a_k \leqslant \beta \leqslant b_k$ et $\forall k \in \mathbb{N}$, $b_k - a_k = \frac{b-a}{2^k}$

On peut alors montrer que si l'entier k est tel que $\frac{b-a}{2^k} \leqslant \varepsilon$, alors a_k et b_k sont des valeurs approchées à ε-près de β.

Une traduction Python du pseudo-code ci-dessus est :

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

3) Le cas d'une loi normale

a) Simulation d'une loi normale centrée réduite $\mathcal{N}(0,1)$.

Le cas d'une loi normale est plus délicat. En effet, dans ce cas, l'expression exacte de la fonction de répartition F est inconnue et encore moins celle de la fonction quantile F^{-1} (il existe bien des tables mais bon...). Nous pouvons cependant utiliser le résultat intéressant suivant pour simuler une loi normale centrée réduite :

Simulation d'une loi normale centrée réduite - Méthode de Box-Mueller

Soit U et V deux variables aléatoires indépendantes de loi uniforme sur $]0,1[$. Alors X et Y définies par :

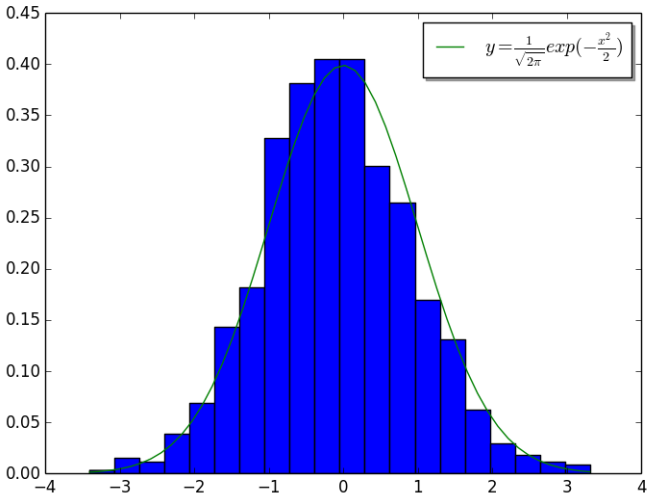
$$X = \sqrt{-2\ln(U)}\cos(2\pi V)$$
$$Y = \sqrt{-2\ln(U)}\sin(2\pi V)$$

sont indépendantes, de loi $\mathcal{N}(0,1)$.

Encore une fois, nous remarquons qu'il suffit de savoir simuler la loi uniforme sur $]0,1[$ pour simuler une loi normale. Ci-dessous, nous présentons une fonction Python simulant la loi $\mathcal{N}(0,1)$.

```
from random import *
from math import *
def NormCentRed():
    u = random()
    v = random()
    return sqrt(-2*log(u))*cos(2*pi*v)
```

et voici un histogramme associé à 1000 réalisations de la loi $\mathcal{N}(0,1)$.




```
Valeurs = np.array([NormCentRed() for k in range(1000)])
plt.hist(Valeurs, bins = 20, normed=1)
X = np.linspace(np.amin(Valeurs),np.amax(Valeurs))
Y = 1/sqrt(2*pi)*np.exp(-X*X/2)
plt.plot(X,Y,label = r'$y=\frac{1}{\sqrt{2\pi}}exp(-\frac{x^2}{2})$')
plt.legend(loc='upper right', shadow=True)
plt.show()
```

b) Simulation d'une loi normale quelconque $\mathcal{N}(\mu,\sigma^2)$.

Soit $(\mu,\sigma)\in\mathbb{R}\times\mathbb{R}_+^*$. Nous savons que si X suit la loi $\mathcal{N}(0,1)$, alors $\mu+\sigma X$ suit la loi $\mathcal{N}(\mu,\sigma^2)$. On déduit donc du code précédent la fonction ci-dessous simulant une variable aléatoire de loi $\mathcal{N}(\mu,\sigma^2)$ où μ et σ sont fournis en entrée de la fonction.

```
from random import *
from math import *
def Normale(moyenne, ecarttype):
    u = random()
    v = random()
    return moyenne + ecarttype*sqrt(-2*log(u))*cos(2*pi*v)
```

4) Utilisation des bibliothèques pour simuler des v.a.

Si on veut aller vite (et plutôt que de tout faire "à la main"), on peut simplement utiliser les fonctions du module random (de la bibliothèque standard) ou ceux du sous-module random du module numpy.

```
from random import *

randint(2,5) # nombre entier au hasard entre 2 et 5 (inclus)
random() # nombre réel au hasard entre 0 et 1
uniform(8,11) # nombre réel au hasard entre 8 et 11
expovariate(3) # simulation d'une loi exponentielle de paramètre 3
normalvariate(3,0.5) # simulation d'une loi normale de moyenne 3, d'écart-type 0.5
normalvariate(0,1) # simulation de la loi normale centrée réduite


import numpy.random as alea

alea.binomial(15,0.3) # simulation d'une loi binomiale avec n = 15 et p = 0.3.
alea.exponential(2.5,size=4) # simulation de 4 valeurs d'une va de loi exp. de paramètre 2.5.
alea.geometric(0.32) # simulation de la loi géométrique de paramètre 0.32.
alea.geometric(0.32,size=5) # idem, mais 5 fois (stocké dans un tableau)
alea.poisson(2.5,size=6) # 6 simulations d'une loi de Poisson de paramètre 2.5

alea.hypergeometric(3,5,2)
'''simulation d'une loi hypergéométrique
(ici de paramètres N = 3+5, n = 2, p = 3/8 = 3/N)'''
```

Attention !

Connaître les fonctions prédéfinies ci-dessus est un plus, mais cela ne dispense pas de maîtriser parfaitement toutes les techniques de simulations exposées précédemment !

Exercices

Simulations de variables aléatoires à densité

Exercice 1 : Simulation d'une v.a.r. suivant la loi de Cauchy

Soit X une variable aléatoire à densité donnée par $f : x \mapsto \frac{1}{\pi} \times \frac{1}{1+x^2}$.

1) Vérifier que f est bien une densité.

2) a) Déterminer la fonction de répartition F de X.

b) Montrer que F réalise une bijection strictement croissante de $\mathbb{R}$ sur $]0,1[$ et déterminer la bijection réciproque F^{-1}.

3) a) Ecrire une fonction Python (sans variable d'entrée) permettant de simuler une réalisation de X.

b) Utiliser cette fonction pour tracer un histogramme représentant 1000 réalisations de X. Faire également apparaître sur ce même graphique la courbe représentative de la densité f.

Exercice 2 : Simulation d'une v.a.r. suivant la loi de Gumbel

Soit X une variable aléatoire à densité donnée par $f : x \mapsto \exp(-x - e^{-x})$.

Reprendre les questions de l'exercice précédent avec cette variable aléatoire.

(*indication* : $\exp(-x - e^{-x}) = e^{-x}\exp(-e^{-x})\ldots$).

Exercice 3 : Simulation d'une v.a.r. suivant la loi de Weibull

Soit X une variable aléatoire de fonction de répartition $F : x \mapsto \begin{cases} 0 \text{ si } x < 0 \\ 1 - e^{-x^2} \text{ si } x \geqslant 0 \end{cases}$ (on dit que X suit une loi de Weibull).

1) a) Montrer que F réalise une bijection de $]0,+\infty[$ sur $]0,1[$ et déterminer F^{-1}.

b) En déduire une fonction Python permettant de simuler X.

2) Écrire une fonction `Python` prenant en entrée $n \in \mathbb{N}^*$ et simulant en sortie la variance empirique $S_n^2 = \frac{X_1^2+\cdots+X_n^2}{n} - \left(\frac{X_1+\cdots+X_n}{n}\right)^2$ d'un n-échantillon de X.

Exercice 4 : Simulation d'une v.a.r. suivant la loi logistique

Soit X une variable aléatoire suivant la loi logistique standard, c'est-à-dire de fonction de répartition $F : x \mapsto \frac{1}{1+e^{-x}}$.

1) Montrer que F réalise une bijection de $\mathbb{R}$ sur $]0,1[$.

2) Écrire une fonction `Python` prenant en entrée $n \in \mathbb{N}^*$ et simulant en sortie la moyenne empirique $\overline{X_n} = \frac{X_1+X_2+\cdots+X_n}{n}$ d'un n-échantillon de X.

Un cas plus délicat

Exercice 5 : Une fonction de répartition bizarre.

1) Vérifier que $F : x \mapsto \frac{\frac{1}{2}+\frac{1}{\pi}\arctan x}{1+e^{-x}}$ est la fonction de répartition d'une v.a. X à densité.
2) Proposer une fonction Python permettant de simuler la variable aléatoire X
(on pourra réécrire la fonction ci-dessus implémentant l'algorithme de dichotomie et y faire appel).

Simulations de lois normales

Exercice 6 : Règle 68

En expérimentant à l'aide de l'ordinateur, vérifier l'assertion ci-dessous :
"Si X suit la loi normale centrée réduite, alors à peu près 68% des valeurs prises par X sont situées dans l'intervalle $[-1, 1]$."

Exercice 7 : Déplacement d'un point mobile dans le plan

Un point mobile se déplace dans un plan muni d'un repère orthonormé. Il part de l'origine O à l'instant 0.
Si à un instant n (où $n \in \mathbb{N}$), il se situe au point de coordonnées (X_n, Y_n), alors à l'instant $n+1$ il se trouve au point de coordonnées (X_{n+1}, Y_{n+1}) de sorte que $A_{n+1} = X_{n+1} - X_n$ et $B_{n+1} = Y_{n+1} - Y_n$ suivent la loi normale centrée réduite.
On suppose que les v.a. A_n et B_m, pour tous $n, m \in \mathbb{N}^*$, sont mutuellement indépendantes.
1) (question mathématique) Déterminer, pour tout entier naturel $n \geqslant 1$, la loi de X_n et la loi de Y_n.
2) Écrire une fonction Python prenant n en entrée ($n \in \mathbb{N}$) et donnant en sortie le trajet du mobile simulé de l'instant 0 à l'instant n.
On présentera le résultat sous forme graphique (ligne brisée reliant les points (X_i, Y_i) pour $0 \leqslant i \leqslant n$).

Loi exponentielle

Exercice 8 : Une illustration du théorème de la limite centrale.

Soit $n \in \mathbb{N}^*$ et $\lambda \in \mathbb{R}_+^*$ et X une variable aléatoire qui suit la loi exponentielle de paramètre λ.
1) Écrire une fonction `Python` prenant n et λ en entrée et donnant en sortie une valeur simulée du couple aléatoire $(\overline{X_n}, S_n^2)$ où

$$\overline{X_n} = \frac{X_1 + X_2 + \cdots + X_n}{n} \text{ et } S_n^2 = \frac{1}{n}\sum_{k=1}^{n}\left(X_k - \overline{X_n}\right)^2$$

sont respectivement les moyenne et variance empiriques d'un n-échantillon de X.
2) On reprend les notations précédentes. Soit $(X_1, X_2, \ldots, X_n)$ un n-échantillon de X.
Écrire une fonction `Python`

- prenant n, λ, un réel $u > 0$ et un entier $m > 0$ en entrée,
- simulant m valeurs de $\frac{\overline{X_n} - \frac{1}{\lambda}}{\frac{S_n}{\sqrt{n}}}$ et donnant en sortie la proportion p des valeurs situées dans l'intervalle $[-u, u]$. (on ne prendra pas en compte le cas -exceptionnel- où éventuellement $S_n = 0$)

3) Pour m grand, on a testé la fonction précédente avec une certaine valeur de u. On obtient des valeurs de p proches de $0,9$. Quel u avons-nous choisi ? Vérifier expérimentalement à l'aide de la fonction précédente.

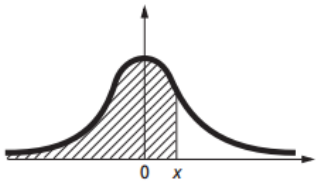
Table 1

Loi normale centrée réduite $\mathcal{N}(0, 1)$

Table de la fonction de répartition

Probabilité d'avoir une valeur inférieure à x :

$$\Pi(x) = P(X \leq x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2} dt$$



x	0,00	0,01	0,02	0,03	0,04	0,05	0,06	0,07	0,08	0,09
0,00	0,5000	0,5040	0,5080	0,5120	0,5160	0,5199	0,5239	0,5279	0,5319	0,5359
0,10	0,5398	0,5438	0,5478	0,5517	0,5557	0,5596	0,5636	0,5675	0,5714	0,5753
0,20	0,5793	0,5832	0,5871	0,5910	0,5948	0,5987	0,6026	0,6064	0,6103	0,6141
0,30	0,6179	0,6217	0,6255	0,6293	0,6331	0,6368	0,6406	0,6443	0,6480	0,6517
0,40	0,6554	0,6591	0,6628	0,6664	0,6700	0,6736	0,6772	0,6808	0,6844	0,6879
0,50	0,6915	0,6950	0,6985	0,7019	0,7054	0,7088	0,7123	0,7157	0,7190	0,7224
0,60	0,7257	0,7291	0,7324	0,7357	0,7389	0,7422	0,7454	0,7486	0,7517	0,7549
0,70	0,7580	0,7611	0,7642	0,7673	0,7704	0,7734	0,7764	0,7794	0,7823	0,7852
0,80	0,7881	0,7910	0,7939	0,7967	0,7995	0,8023	0,8051	0,8078	0,8106	0,8133
0,90	0,8159	0,8186	0,8212	0,8238	0,8264	0,8289	0,8315	0,8340	0,8365	0,8389
1,00	0,8413	0,8438	0,8461	0,8485	0,8508	0,8531	0,8554	0,8577	0,8599	0,8621
1,10	0,8643	0,8665	0,8686	0,8708	0,8729	0,8749	0,8770	0,8790	0,8810	0,8830
1,20	0,8849	0,8869	0,8888	0,8907	0,8925	0,8944	0,8962	0,8980	0,8997	0,9015
1,30	0,9032	0,9049	0,9066	0,9082	0,9099	0,9115	0,9131	0,9147	0,9162	0,9177
1,40	0,9192	0,9207	0,9222	0,9236	0,9251	0,9265	0,9279	0,9292	0,9306	0,9319
1,50	0,9332	0,9345	0,9357	0,9370	0,9382	0,9394	0,9406	0,9418	0,9429	0,9441
1,60	0,9452	0,9463	0,9474	0,9484	0,9495	0,9505	0,9515	0,9525	0,9535	0,9545

Il est également possible de simuler une variable aléatoire suivant une loi de Poisson de paramètre λ à partir de variables aléatoires indépendantes de loi exponentielles de paramètre λ.
Le résultat est le suivant (l'exercice 10 en propose une démonstration) :
Soit $X_1, X_2, \ldots, X_n, \ldots$ des variables aléatoires indépendantes de même loi exponentielle de paramètre λ. La variable aléatoire N définie par :

$$N = 0 \text{ si } X_1 > 1 \text{ et } N = \max_{i \geqslant 1} \left\{ \sum_{j=1}^{i} X_j \leqslant 1 \right\} \text{ sinon}$$

suit la loi de Poisson de paramètre λ.

Exercice 9 : simulation d'une loi de Poisson

On reprend les notations du résultat précédent (que l'on admet).
On note $U_1, U_2, \ldots, U_n, \ldots$ des variables aléatoires indépendantes de loi uniforme sur $]0, 1[$.
1) Montrer :

- $P(N = 0) = P(U_1 < e^{-\lambda})$.
- si $n \geqslant 1$, $P(N = n) = P\left(\prod_{j=1}^{n} U_j \geqslant e^{-\lambda} > \prod_{j=1}^{n+1} U_j \right)$.

2) En déduire une fonction Python permettant de simuler une v.a. qui suit la loi $\mathcal{P}(\lambda)$.

Exercice 10 (mathématique)

Soit $\lambda > 0$. On considère une suite X_1, X_2,..., X_n,... de variables aléatoires indépendantes de même loi exponentielle de paramètre λ et, pour tout entier naturel n, on pose $S_n = X_1 + X_2 + \cdots + X_n$.

1) Montrer que S_n est une variable aléatoire à densité f_{S_n} donnée par :

$$f_{S_n}(x) = \frac{e^{-\lambda x}\lambda^n x^{n-1}}{(n-1)!}\mathbb{1}_{]0,+\infty[}(x)$$

2) On pose N la variable aléatoire discrète définie par :

$$N = 0 \text{ si } X_1 > 1 \text{ et } N = \max_{i\geqslant 1}\left\{\sum_{j=1}^{i} X_j \leqslant 1\right\} \text{ sinon}$$

L'objectif est de déterminer la loi de N.

a) Exprimer à l'aide de la variable aléatoire S_n l'évènement $(N \geqslant n)$.

b) Etablir $\forall n \in \mathbb{N}$, $1 = \sum_{j=0}^{n} e^{-\lambda}\frac{\lambda^j}{j!} + \int_0^\lambda e^{-t}\frac{t^n}{n!}\mathrm{d}t$ et en déduire $P(S_{n+1} \leqslant 1) = 1 - \sum_{j=0}^{n} e^{-\lambda}\frac{\lambda^j}{j!}$.

c) Identifier la fonction de répartition de N puis reconnaître la loi de N.

Corrigés :

Exercice 1 *Corrigé* :

1) f est clairement continue et positive sur $\mathbb{R}$. De plus, si $A > 0$,
$\int_0^A f(t)\mathrm{d}t = \frac{1}{\pi}(\arctan A - \arctan 0) = \frac{1}{\pi}\arctan A$ donc $\lim\limits_{A\to+\infty}\int_0^A f(t)\mathrm{d}t = \frac{1}{\pi}\times\frac{\pi}{2} = \frac{1}{2}$.
Ainsi, $\int_0^{+\infty} f(t)\mathrm{d}t = \frac{1}{2}$. De même, si $B < 0$, $\int_B^0 f(t)\mathrm{d}t = -\frac{1}{\pi}\arctan B$ donc
$\lim\limits_{B\to-\infty}\int_B^0 f(t)\mathrm{d}t = -\frac{1}{\pi}\times\left(-\frac{\pi}{2}\right) = \frac{1}{2}$. Ainsi $\int_{-\infty}^0 f(t)\mathrm{d}t = \frac{1}{2}$. Finalement, $\int_{-\infty}^{+\infty} f(t)\mathrm{d}t$ converge et
$\int_{-\infty}^{+\infty} f(t)\mathrm{d}t = 1$. Conclusion : $\boxed{f \text{ est bien une densité}}$.

2) a) Pour tout $x \in \mathbb{R}$, $F(x) = \int_{-\infty}^x f(t)\mathrm{d}t = \lim\limits_{B\to-\infty}\int_B^x f(t)\mathrm{d}t = \frac{1}{\pi}\lim\limits_{B\to-\infty}[\arctan t]_B^x = \boxed{\frac{1}{\pi}(\arctan x + \frac{\pi}{2})}$.

b) F est clairement strictement croissante et continue sur $\mathbb{R}$ et $F(\mathbb{R}) =]0,1[$
(car $\lim\limits_{x\to-\infty}\frac{1}{\pi}(\arctan x + \frac{\pi}{2}) = 0$ et $\lim\limits_{x\to+\infty}\frac{1}{\pi}(\arctan x + \frac{\pi}{2}) = \frac{1}{\pi}(\frac{\pi}{2} + \frac{\pi}{2}) = 1$) donc, d'après le théorème de la bijection, F réalise une bijection de $\mathbb{R}$ sur $]0,1[$. Déterminons la bijection réciproque $F^{-1} :]0,1[\to \mathbb{R}$. Soit $y \in]0,1[$ et $x \in \mathbb{R}$.

$$\begin{aligned} F(x) = y &\Leftrightarrow \frac{1}{\pi}(\arctan x + \frac{\pi}{2}) = y \Leftrightarrow \arctan x = \pi\left(y - \frac{1}{2}\right) \\ &\Leftrightarrow x = \tan\left(\pi\left(y - \frac{1}{2}\right)\right) \ \left(\text{car } \pi\left(y - \frac{1}{2}\right) \in \left]-\frac{\pi}{2}, \frac{\pi}{2}\right[\right) \end{aligned}$$

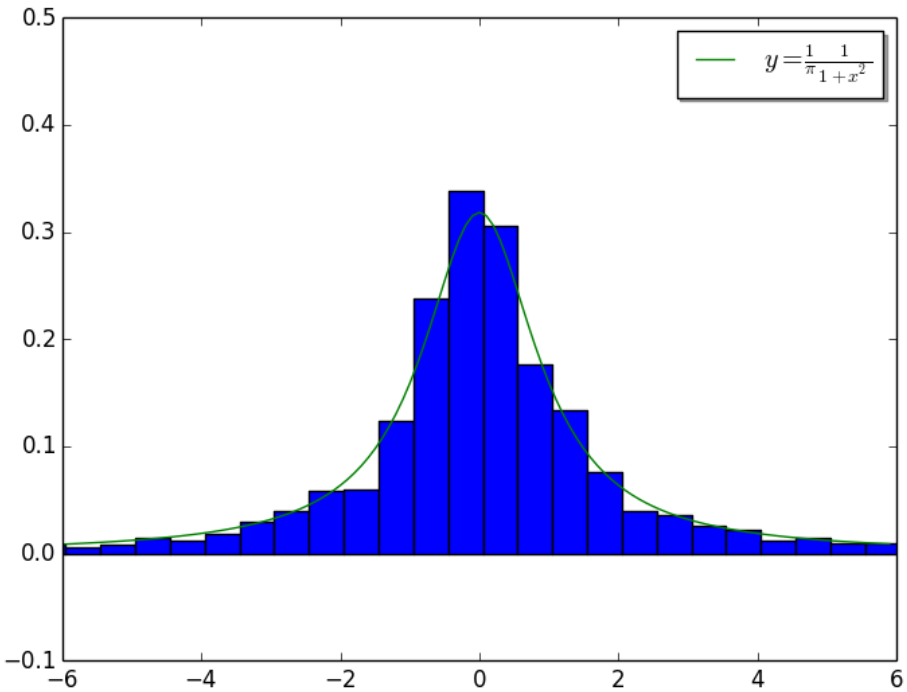
Ainsi, $\boxed{F^{-1}(y) = \tan\left(\pi\left(y - \frac{1}{2}\right)\right)}$.

3) a) On utilise le fait que, si U suit une loi uniforme sur $]0,1[$, alors $F^{-1}(U)$ suit la loi de Cauchy.

```
from random import *
from math import *

def Cauchy():
    u = random()
    return tan(pi*(u-1/2))
```

b) On s'inspire du code fourni pour la loi exponentielle avec un ajustement de la fenêtre graphique nécessaire ici (on limite les abscisses de -6 à 6 et les ordonnées de -0.1 à 0.5).

```
import numpy as np
import matplotlib.pyplot as plt

Valeurs = np.array([Cauchy() for k in range(1000)]) # 1000 simulations de X
plt.hist(Valeurs, bins = 400, normed=1) # tracé de l'histogramme
plt.axis(xmin = -6, xmax = 6, ymin = -0.1, ymax = 0.5)
X = np.arange(-6,6,0.1) # abscisses: subdivision de [-6,6] avec un pas de 0.1
Y = 1/pi*1/(1+X*X)  # ordonnées des points de la densité
plt.plot(X,Y,label = r'$y=\frac{1}{\pi}\frac{1}{1+x^2}$') # tracé de la densité
plt.legend(loc='upper right', shadow=True) # légende (en haut à droite)
plt.show()
```



Exercice 3 *Corrigé* :
1) a) F est dérivable sur $]0,+\infty[$, de dérivée $F' : x \mapsto 2xe^{-x^2}$. Ainsi, $\forall x > 0$, $F'(x) > 0$ et F réalise une bijection strictement croissante de $]0,+\infty[$ sur $F(]0,+\infty[) =]0,1[$ (car $\lim\limits_{0} F = F(0) = 0$ et $\lim\limits_{+\infty} F = 1$).
Pour $y \in]0,1[$ et $x \in]0,+\infty[$,

$$\begin{aligned} y = F(x) &\Leftrightarrow y = 1 - e^{-x^2} \Leftrightarrow e^{-x^2} = 1 - y \Leftrightarrow -x^2 = \ln(1-y) \\ &\Leftrightarrow x^2 = -\ln(1-y) = \ln\left(\frac{1}{1-y}\right) \\ &\Leftrightarrow x = \sqrt{\ln\left(\frac{1}{1-y}\right)} \text{ (en effet, ici } x > 0 \text{ et } \ln\left(\frac{1}{1-y}\right) > 0 \text{ car } \frac{1}{1-y} > 1). \end{aligned}$$

Ainsi, $F^{-1}(y) = \sqrt{\ln\left(\frac{1}{1-y}\right)}$ pour tout $y \in]0,1[$.
b) On utilise le fait que, si U suit une loi uniforme sur $]0,1[$, alors $F^{-1}(U)$ suit une loi de Weibull (qui est aussi ici une loi de Rayleigh). On remarque aussi que U et $1-U$ suivent la même loi ce qui permet de remplacer $1-U$ par U dans l'expression de $F^{-1}(U)$.

```
from random import *
from math import sqrt, log

def Weibull():
    U = random()
    return sqrt(log(1/U))
```

2) Il suffit d'utiliser la fonction précédente et de savoir calculer des moyennes.

```
def VarWeibull(n):
    SommeCar, Somme = 0, 0
    for k in range(n):
        x = Weibull()
        Somme += x
        SommeCar += x**2
    return SommeCar/n - (Somme/n)**2
```

Exercice 9 *Corrigé* :
On sait que $-\frac{\ln(U_1)}{\lambda}$, $-\frac{\ln(U_2)}{\lambda}$, ..., $-\frac{\ln(U_n)}{\lambda}$, ... sont des variables aléatoires indépendantes suivant chacune une loi exponentielle de paramètre λ.
En posant $X_i = -\frac{\ln(U_i)}{\lambda}$ pour tout entier naturel i, on peut donc utiliser le résultat :
La variable aléatoire N définie par :

$$N = 0 \text{ si } X_1 > 1 \text{ et } N = \max_{i \geqslant 1} \left\{ \sum_{j=1}^{i} X_j \leqslant 1 \right\} \text{ sinon}$$

suit la loi de Poisson de paramètre λ.
1) On a

$$\begin{aligned}(N=0)=(X_1>1) &= \left(-\frac{\ln(U_1)}{\lambda}>1\right)=\left(\frac{\ln(U_1)}{\lambda}<-1\right)=(\ln(U_1)<-\lambda)\\ &= (U_1<\exp(-\lambda))\,.\end{aligned}$$

donc $P(N=0)=P(U_1<e^{-\lambda})$.
Pour $n\geqslant 1$,

$$\begin{aligned}(N=n) &= \left(\max_{i\geqslant 1}\left\{\sum_{j=1}^{i}X_j\leqslant 1\right\}=n\right)=\left(\sum_{j=1}^{n}X_j\leqslant 1<\sum_{j=1}^{n+1}X_j\right)\\ &= \left(-\frac{1}{\lambda}\sum_{j=1}^{n}\ln(U_j)\leqslant 1<-\frac{1}{\lambda}\sum_{j=1}^{n+1}\ln(U_j)\right)\\ &= \left(\sum_{j=1}^{n}\ln(U_j)\geqslant -\lambda>\sum_{j=1}^{n+1}\ln(U_j)\right)\\ &= \left(\exp\left(\sum_{j=1}^{n}\ln(U_j)\right)\geqslant e^{-\lambda}>\exp\left(\sum_{j=1}^{n+1}\ln(U_j)\right)\right)\\ &= \left(\prod_{j=1}^{n}\exp(\ln(U_j))\geqslant e^{-\lambda}>\prod_{j=1}^{n+1}\exp(\ln(U_j))\right)\\ &= \left(\prod_{j=1}^{n}U_j\geqslant e^{-\lambda}>\prod_{j=1}^{n+1}U_j\right)\end{aligned}$$

donc $P(N=n)=P\left(\prod_{j=1}^{n}U_j\geqslant e^{-\lambda}>\prod_{j=1}^{n+1}U_j\right)$.
2) Méthode : On simule N. Pour cela, on simule $\prod_{j=1}^{n}U_j$ pour $n=1,2,\ldots$ tant que $\prod_{j=1}^{n}U_j\geqslant e^{-\lambda}$ et on renvoie en sortie le dernier n qui a satisfait cette condition (et 0 si on est sorti de la boucle dès le début de celle-ci).

```python
from random import *
from math import exp

def Poisson(lambd):
    Prod = random()
    m = 1
    while Prod >= exp(-lambd):
        Prod *= random()
        m += 1
    return m-1
```

Corrigés d'exercices sur les simulations de lois à densité

Exercice 5 : Une fonction de répartition bizarre.

1) Vérifier que $F : x \mapsto \frac{\frac{1}{2}+\frac{1}{\pi}\arctan x}{1+e^{-x}}$ est la fonction de répartition d'une v.a. X à densité.
2) Proposer une fonction Python permettant de simuler la variable aléatoire X
(on pourra réécrire la fonction ci-dessus implémentant l'algorithme de dichotomie et y faire appel).

Corrigé :
1) F est une fonction définie et de classe C^1 sur $\mathbb{R}$. De plus $\lim\limits_{-\infty} F = 0$ et $\lim\limits_{+\infty} F = \frac{\frac{1}{2}+\frac{1}{\pi}\times\frac{\pi}{2}}{1} = 1$.
Enfin, $\forall x \in \mathbb{R}$, $F'(x) = \frac{\frac{1}{\pi(1+x^2)}(1+e^{-x})+\left(\frac{1}{2}+\frac{1}{\pi}\arctan x\right)e^{-x}}{(1+e^{-x})^2} > 0$.
Ainsi, F' est continue positive et $\int_{-\infty}^{+\infty} F'(x)\mathrm{d}x = \lim\limits_{+\infty} F - \lim\limits_{-\infty} F = 1$.
On en déduit que F' est la densité d'une variable aléatoire X dont la fonction de répartition est F.
2) Bien que F est réalise une bijection croissante de $\mathbb{R}$ sur $]0,1[$,
il apparaît ici très délicat d'expliciter F^{-1}.
Pour simuler X, on va donc choisir aléatoirement un réel u dans l'intervalle $]0,1[$ et donner une valeur approchée de la solution $x \in \mathbb{R}$ de $F(x) - u = 0$ en procédant par dichotomie (avec une précision ε égale à 10^{-6} par exemple). On choisira $a = -10$ et $b = 10$ dans la méthode de dichotomie (sachant que $F(-10) \simeq 1,44 \times 10^{-6}$ et $F(10) \simeq 0,97$.
Pour s'assurer que $u \in [a,b]$, on va modifier préalablement les bornes a et b si nécessaire.
Si $F(a) - u \geqslant 0$, on multipliera a par 2, si $F(b) - u \leqslant 0$, on multipliera b par 2. On répètera cela autant que nécessaire pour que, finalement, $F(a) - u < 0$ et $F(b) - u > 0$.

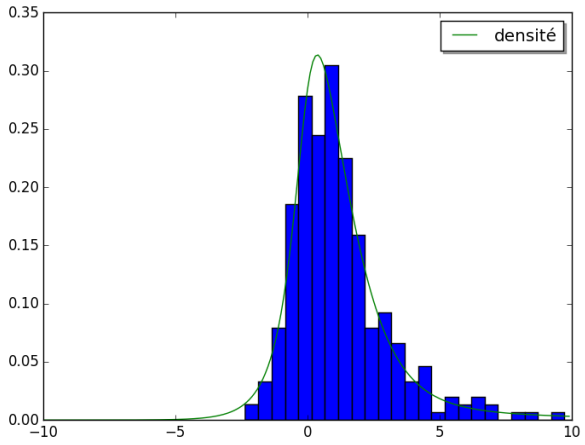
```python
from random import *
from math import *
import matplotlib.pyplot as plt

def fdicho(x,u):
    return (1/2+1/pi*atan(x))/(1+exp(-x)) - u

def dichotomie(f,a,b,u,eps):
    while (b-a) > eps:
        c = (a + b)/2
        if f(a,u)*f(c,u) <= 0:
            b = c
        else:
            a = c
    return (a+b)/2

def Simul():
    a = -10
    b = 10
    epsilon = 10**(-6)
    u = random()
    while fdicho(a,u) >= 0:
        a *= 2
    while fdicho(b,u) <= 0:
        b *= 2
    return dichotomie(fdicho,a,b,u,epsilon)
```

Pour vérifier ce programme, on peut superposer une densité de la loi avec un histogramme des valeurs prises par la variable aléatoire. Ici, nous avons déjà remarqué qu'une densité f était donnée par :

$$\forall x \in \mathbb{R},\ f(x) = F'(x) = \frac{\frac{1}{\pi(1+x^2)}(1+e^{-x}) + \left(\frac{1}{2} + \frac{1}{\pi}\arctan x\right)e^{-x}}{(1+e^{-x})^2}.$$

```
import numpy as np

def Densite(x):
    return 1/(pi*(1+x**2))/(1+exp(-x))+(1/2+1/pi*atan(x))*exp(-x)/(1+exp(-x))**2

Valeurs = np.array([Simul() for k in range(300)]) # 300 simulations de X
plt.hist(Valeurs, bins = 100, normed=1) # tracé de l'histogramme
X = np.arange(-10,10,0.1)
Y = np.vectorize(Densite)(X)  # ordonnées des points de la densité
plt.plot(X,Y,label = "densité") # tracé de la densité
plt.axis(xmin=-10,xmax=10,ymin=0,ymax=0.35)
plt.legend(loc='upper right', shadow=True) # légende (en haut à droite)
plt.show()
```

(l'instruction `np.vectorize(Densite)` permet de "vectoriser" la fonction `Densite`, c'est-à-dire de l'appliquer à un vecteur créé avec `numpy`)
On obtient par exemple le graphique (où l'histogramme semble bien cohérent avec la densité de probabilité) :



Simulations de lois normales

Exercice 6 : Règle 68

En expérimentant à l'aide de l'ordinateur, vérifier l'assertion ci-dessous :
"Si X suit la loi normale centrée réduite, alors à peu près 68% des valeurs prises par X sont situées dans l'intervalle $[-1,1]$."

Corrigé :
Corrigé : On peut créer une fonction pourcentage qui prend en entrée un entier naturel n et qui simule n réalisation d'une loi normale centrée réduite. La fonction donne en sortie le pourcentage de réalisations situées dans l'intervalle $[-1,1]$.

```
1 def pourcentage(n):
2     nbvaleurs = 0
3     for k in range(n):
4         if -1 <= NormCentRed() <= 1:
5             nbvaleurs += 1
6     return 100*nbvaleurs/n
```

On peut alors appeler la fonction pour diverses valeurs du paramètre n.

```
In [32]: pourcentage(100)
Out[32]: 65.0

In [33]: pourcentage(1000)
Out[33]: 69.4

In [34]: pourcentage(10000)
Out[34]: 68.29

In [35]: pourcentage(100000)
Out[35]: 68.226
```

Et l'on constate que l'assertion est d'autant plus pertinente que le nombre de simulations n est grand.

Exercice 7 : Déplacement d'un point mobile dans le plan

Un point mobile se déplace dans un plan muni d'un repère orthonormé. Il part de l'origine O à l'instant 0.
Si à un instant n (où $n \in \mathbb{N}$), il se situe au point de coordonnées (X_n, Y_n), alors à l'instant $n+1$ il se trouve au point de coordonnées (X_{n+1}, Y_{n+1}) de sorte que $A_{n+1} = X_{n+1} - X_n$ et $B_{n+1} = Y_{n+1} - Y_n$ suivent la loi normale centrée réduite.
On suppose que les v.a. A_n et B_m, pour tous $n, m \in \mathbb{N}^*$, sont mutuellement indépendantes.
1) (question mathématique) Déterminer, pour tout entier naturel $n \geqslant 1$, la loi de X_n et la loi de Y_n.
2) Écrire une fonction Python prenant n en entrée ($n \in \mathbb{N}$) et donnant en sortie le trajet du mobile simulé de l'instant 0 à l'instant n.
On présentera le résultat sous forme graphique (ligne brisée reliant les points (X_i, Y_i) pour $0 \leqslant i \leqslant n$).

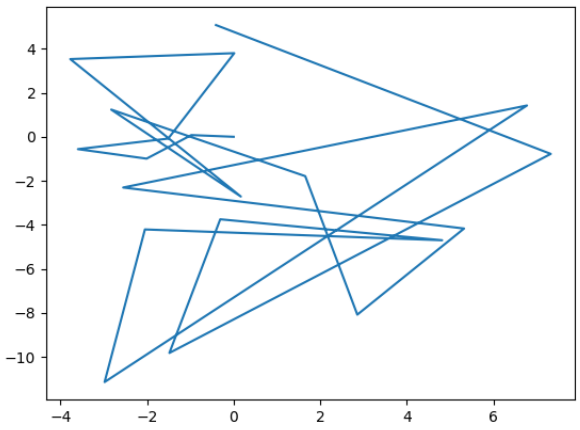
Corrigé :
1) Soit $n \in \mathbb{N}^*$. Comme $(X_0 = 0)$ et $(Y_0 = 0)$ sont des évènements certains, on a par télescopage

$$X_n = \sum_{k=0}^{n-1}(X_{k+1} - X_k) = \sum_{k=0}^{n-1} A_{k+1} \text{ et } Y_n = \sum_{k=0}^{n-1}(Y_{k+1} - Y_k) = \sum_{k=0}^{n-1} B_{k+1}.$$

Ainsi, X_n apparaît comme une somme de n variables aléatoires indépendantes de loi commune $\mathcal{N}(0,1)$ donc X_n suit la loi $\mathcal{N}(0,n)$. Pour les mêmes raisons, Y_n suit la loi $\mathcal{N}(0,n)$.

2) On se sert du fait que, pour tout $k \in [\![0,k]\!]$, X_k et Y_k suivent la loi $\mathcal{N}(0,k)$ et que cette loi peut être simulée avec la méthode de Box-Mueller (si U,V sont deux variables aléatoires indépendantes de loi uniforme sur $]0,1[$, alors $\sqrt{k}\times\sqrt{-2\ln(U)}\cos(2\pi V)$ suit la loi $\mathcal{N}(0,k)$).

```
from matplotlib import pyplot as plt
from random import *
from math import log, sqrt, cos, pi

def Trajet(n):
    abs, ord = [0], [0]
    for k in range(1,n+1):
        u, v = random(), random()
        uu, vv = random(), random()
        x = sqrt(-2*k*log(u))*cos(2*pi*v)
        y = sqrt(-2*k*log(uu))*cos(2*pi*vv)
        abs.append(x)
        ord.append(y)
    plt.plot(abs,ord)
    plt.show()
```



Analyse numérique élémentaire

Cours d'analyse numérique élémentaire

Quelques méthodes rudimentaires d'analyse numérique

1) Méthodes de Newton et de dichotomie pour la recherche du zéro d'une fonction sur un intervalle donné $[a,b]$

Pour la méthode de dichotomie, on suppose la fonction f continue, strictement monotone sur $[a,b]$, telle que 0 appartient à l'intervalle d'extrémités $f(a)$ et $f(b)$. On veut déterminer l'unique solution x de $f(x)=0$ sur l'intervalle $[a,b]$.

Idée :
Si $f(a)$ et $f(b)$ sont de signes contraires avec $a \leqslant b$, alors f s'annule sur $[a,b]$. Si $f(a)$ et $f\left(\frac{a+b}{2}\right)$ sont de signes contraires, alors f s'annule sur $\left[a, \frac{a+b}{2}\right]$ sinon f s'annule sur $\left[\frac{a+b}{2}, b\right]$. Dans les deux cas, on obtient un intervalle deux fois plus petit contenant le zéro de f sur $[a,b]$. On poursuit ainsi avec ce nouvel intervalle jusqu'à obtenir la précision souhaitée.

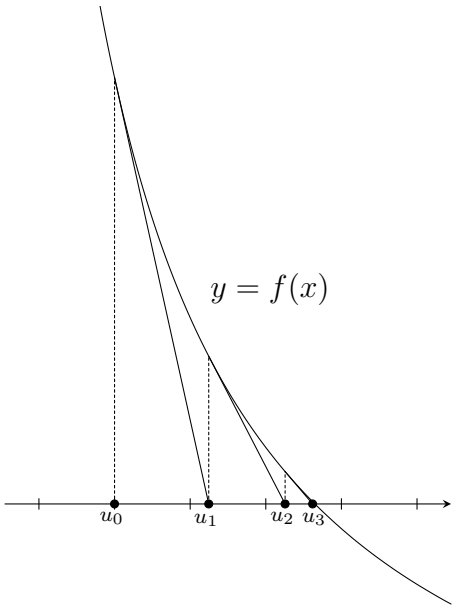
```
def dichotomie(f,a,b,eps):
    while (b-a) > eps:
        c = (a + b)/2
        if f(a)*f(c) <= 0:
            b = c
        else:
            a = c
    return (a+b)/2
```

Pour la méthode de Newton,

❶ Résumé de la méthode : il s'agit de calculer les termes de la suite (u_n) (si bien définie) pour obtenir une valeur approchée d'une solution sur $[a,b]$ de l'équation $f(x)=0$:

$$\text{On choisit judicieusement } u_0 \text{ et } \forall n \in \mathbb{N},\ u_{n+1} = u_n - \frac{f(u_n)}{f'(u_n)}.$$

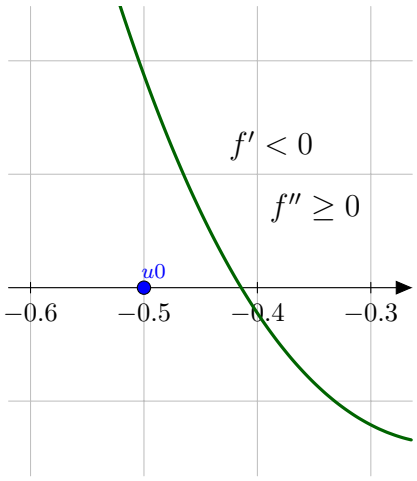
Remarque : L'équation de la tangente au point d'abscisse u_n est $y = f(u_n)+f'(u_n)(x-u_n)$. Elle coupe l'axe des abscisses au point d'ordonnée $y=0$ et l'abscisse x vérifie alors $f(u_n)+f'(u_n)(x-u_n)=0$, c'est-à-dire $x-u_n = -\frac{f(u_n)}{f'(u_n)}$ ou encore $x = u_n - \frac{f(u_n)}{f'(u_n)}$. Ainsi, u_{n+1} est l'abscisse du point d'intersection de l'axe des abscisses avec la tangente à la courbe de f au point d'abscisse u_n.

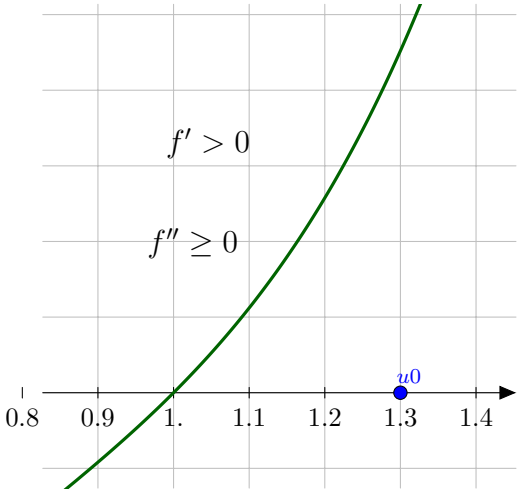


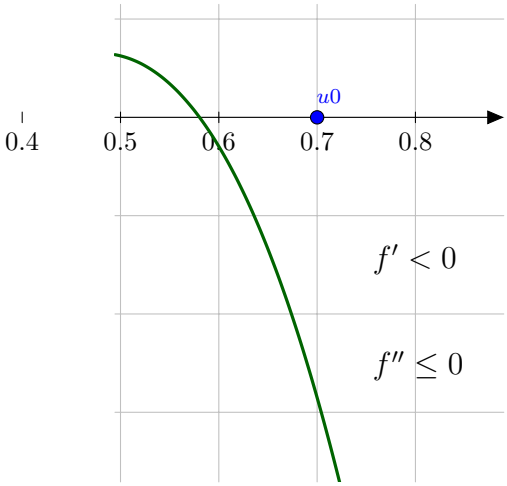
❷ Code Python de l'algorithme de Newton pour obtenir une valeur approchée d'une solution sur $[a,b]$ de l'équation $f(x)=0$:

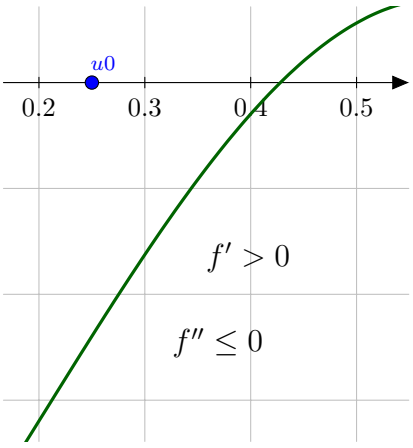
```
''' f est la fonction et df sa dérivée
(à définir au préalable)'''
def Newton(u0,n):
        U = u0
        for k in range(n):
                U = U - f(U)/df(U)
        return U
```

❸ Cas favorables d'application de cet algorithme (répertoriés ci-dessous, avec un choix judicieux de u_0 par rapport au zéro de f) :









Remarques

- La méthode de dichotomie est à préférer si l'on dispose de peu d'hypothèses sur f. De plus, elle permet d'obtenir une approximation du point où s'annule f avec une précision ε choisie par l'utilisateur.
- Dans les cas favorables, la méthode de Newton donne cependant une valeur approchée très précise

de la solution de $f(x) = 0$ avec seulement une dizaine d'itérations (et elle est très simple à mettre en oeuvre).

Exercice 1

Pour $n \in \mathbb{N}^*$, on considère le polynôme $P_n = X^n - (1-X)^3$.

1) Montrer que, pour tout entier naturel $n \geqslant 1$, le polynôme P_n possède dans l'intervalle $]0,1[$ une unique racine que l'on notera α_n.

2) a) Écrire une fonction Python prenant en entrée n et donnant en sortie une valeur approchée de α_n à 10^{-6} près.

b) En déduire la liste des 20 premières valeurs de $(\alpha_n)_{n\geqslant 1}$. Que conjecturer ?

c) Déterminer la liste $[\alpha_{1000}, \alpha_{2000}, \alpha_{3000}, \ldots, \alpha_{10000}]$. Que conjecturer ?

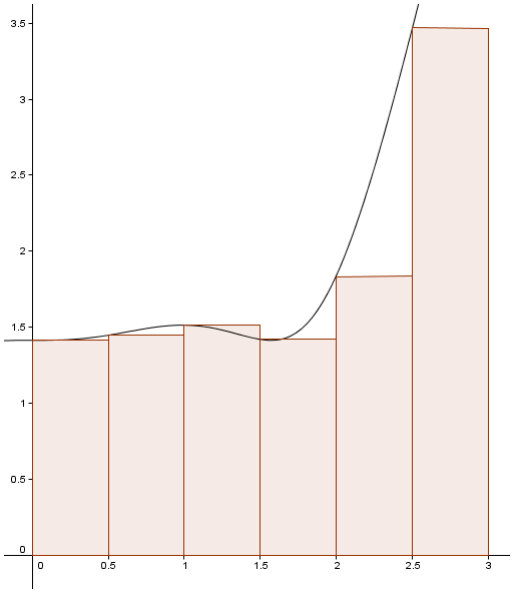
2) Calcul approché d'une intégrale $\int_a^b f(x)\mathrm{d}x$

La méthode des rectangles.
Soit f une fonction continue sur $[a,b]$. On rappelle la formule des rectangles à gauche :

$$\int_a^b f(t)\mathrm{d}t = \lim_{N\to+\infty} \frac{b-a}{N}\sum_{k=0}^{N-1} f\left(a+k\frac{b-a}{N}\right).$$

La formule des rectangles à droite est :

$$\int_a^b f(t)\mathrm{d}t = \lim_{N\to+\infty} \frac{b-a}{N}\sum_{k=1}^{N} f\left(a+k\frac{b-a}{N}\right).$$



Remarques

Les N rectangles ont tous la même largeur égale au pas $\frac{b-a}{N}$ de la subdivision de $[a,b]$.

- Si les N rectangles sont numérotés à partir de 0, alors $a+k\frac{b-a}{N}$ est l'abscisse du bord gauche du k-ième rectangle.
- Si les N rectangles sont numérotés à partir de 1, alors $a+k\frac{b-a}{N}$ est l'abscisse du bord gauche du k-ième rectangle.

Codes des deux méthodes :

```
def RectangleGauche(f,a,b,N):
    Pas = (b-a)/N
    Somme = 0
    for k in range(N):
        Somme += f(a+k*Pas)
    return Pas*Somme
```

```
def RectangleDroite(f,a,b,N):
    Pas = (b-a)/N
    Somme = 0
    for k in range(1,N+1):
        Somme += f(a+k*Pas)
    return Pas*Somme
```

Exercice 2

Soit a,b deux réels tels que $a<b$ et $f:[a,b]\to\mathbb{R}$ une fonction continue et croissante. L'objectif de cet exercice est de démontrer que les suites de sommes de Riemann

$$\left(\frac{b-a}{n}\sum_{k=0}^{n-1} f\left(a+k\frac{b-a}{n}\right)\right)_{n\geqslant 1} \quad\text{et}\quad \left(\frac{b-a}{n}\sum_{k=1}^{n} f\left(a+k\frac{b-a}{n}\right)\right)_{n\geqslant 1}$$

convergent bien vers $\int_a^b f(t)\mathrm{d}t$ et d'avoir une majoration de l'erreur commise par ces approximations.

Pour tout entier naturel n non nul, on pose

$$U_n = \frac{b-a}{n}\sum_{k=0}^{n-1} f\left(a+k\frac{b-a}{n}\right) \text{ et } V_n = \frac{b-a}{n}\sum_{k=1}^{n} f\left(a+k\frac{b-a}{n}\right).$$

1) Soit $n \in \mathbb{N}^*$.

a) Montrer : $\forall k \in [\![1,n]\!]$, $\forall x \in \left[a+(k-1)\frac{b-a}{n}, a+k\frac{b-a}{n}\right]$,

$$\frac{b-a}{n} f\left(a+(k-1)\frac{b-a}{n}\right) \leqslant \int_{a+(k-1)\frac{b-a}{n}}^{a+k\frac{b-a}{n}} f(x)\mathrm{d}x \leqslant \frac{b-a}{n} f\left(a+k\frac{b-a}{n}\right).$$

b) En déduire :

$$\forall n \in \mathbb{N}^*,\ U_n \leqslant \int_a^b f(x)\mathrm{d}x \leqslant V_n$$

2) Calculer et simplifier $V_n - U_n$. En déduire :

$$\forall n \in \mathbb{N}^*,\ 0 \leqslant \int_a^b f(x)\mathrm{d}x - U_n \leqslant \frac{b-a}{n}(f(b)-f(a)).$$

3) **Applications**

a) Écrire une fonction Python qui, pour une fonction f continue et croissante sur un intervalle $[a,b]$, donne en sortie une approximation de $\int_a^b f(t)\mathrm{d}t$ à ε-près où f, a, b, ε seront fournis en entrée de la fonction (avec $\varepsilon \in \mathbb{R}_+^*$).

b) En déduire une approximation à 10^{-5} près des intégrales $\int_2^6 \frac{e^t}{t^2}\mathrm{d}t$, $\int_0^2 e^x \arctan(x)\mathrm{d}x$ et $\int_4^{10} \sqrt{1+\arctan(x)\ln(x+2)}\mathrm{d}x$.

3) Méthode d'Euler pour une valeur approchée de $\varphi(\beta)$ où φ est solution de $y' = f(t,y(t))$ sur $[\alpha,\beta]$ avec $\varphi(\alpha) = y_0$ et un pas $h = \frac{\beta-\alpha}{N}$

Principe :

Considérons une équation différentielle $(E) : y' = f(t,y)$ du premier ordre avec $f :]a,b[\times \mathbb{R} \to \mathbb{R}$ une fonction de deux variables. Supposons qu'il existe une solution φ de (E) définie sur un segment $[\alpha,\beta]$.

On pose $y_0 = \varphi(\alpha)$. Comme $\varphi'(\alpha) = \lim\limits_{h\to 0} \frac{\varphi(\alpha+h)-\varphi(\alpha)}{h}$, on a pour h très petit $\frac{\varphi(\alpha+h)-\varphi(\alpha)}{h} \simeq \varphi'(\alpha) = f(\alpha,\varphi(\alpha))$ donc $\varphi(\alpha+h)-\varphi(\alpha) \simeq h \times f(\alpha,\varphi(\alpha))$ donc

$$\varphi(\alpha+h) \simeq \varphi(\alpha) + h \times f(\alpha,\varphi(\alpha)).$$

Pour obtenir une valeur approchée y_1 de $\varphi(\alpha+h)$, on remplace $\varphi(\alpha)$ par y_0 dans l'approximation précédente. On définit ainsi y_1 par $\boxed{y_1 = y_0 + hf(\alpha,y_0)}$.

De même,

$$\varphi(\alpha+2h) \simeq \varphi(\alpha+h) + h \times f(\alpha+h,\varphi(\alpha+h)).$$

Cette fois, on remplace $\varphi(\alpha+h)$ par y_1 et on pose $\boxed{y_2 = y_1 + h \times f(\alpha+h,y_1)}$. On remarque que y_2 est une approximation de $\varphi(\alpha+2h)$ calculée à partir d'une approximation de $\varphi(\alpha+h)$ (à savoir y_1).

Plus généralement, on définit par récurrence y_k par :

$$y_0 = \varphi(\alpha) \text{ et } \ y_{k+1} = y_k + h \times f(\alpha+h\times k, y_k)$$

de sorte que y_k soit une valeur approchée de $\varphi(\alpha+h\times k)$. Les valeurs approchées $y_0, y_1, y_2, \ldots$ sont de plus en plus mauvaises car elles s'appuient sur les approximations déjà effectuées précédemment, d'où l'intérêt de choisir h très petit.

Point méthode

On choisit un pas $h>0$ suffisamment petit pour définir ce qu'on appelle *une subdivision* $\{\alpha, \alpha+h, \alpha+2h, \ldots, \beta\}$ du segment $[\alpha,\beta]$. Pour cela, on choisit un grand entier naturel N et on pose $h=\frac{\beta-\alpha}{N}$ (de sorte que $\alpha+Nh=\beta$).
Pour avoir une valeur approchée de $\varphi(\alpha+hk)$ (avec $k\in[\![0,N]\!]$), on calcule le terme général y_k de la suite finie définie par :

$$\begin{cases} y_0=\varphi(\alpha) \\ \text{et la relation de récurrence :} \\ \forall k\in[\![0,N-1]\!],\ y_{k+1}=y_k+h\times f(\alpha+h\times k, y_k) \end{cases}$$

Si $h=\frac{\beta-\alpha}{N}$ avec N très grand, alors h sera très petit et $y_N\simeq\varphi(\alpha+h\times N)=\varphi(\beta)$ d'où le code ci-dessous de la méthode d'Euler pour obtenir une valeur approchée de $\varphi(\beta)$ telle que $\varphi(\alpha)=u$ (où u est fourni par l'utilisateur de la fonction).

```
def Euler(f,alpha,beta,u,N):
        y = u
        h = (beta - alpha)/N
        for k in range(N):
                y = y + h*f(alpha + h*k, y)
        return y
```

On peut aussi stocker les valeurs $\alpha, \alpha+h, \alpha+2h, \ldots, \alpha+Nh=\beta$ et $y_0, y_1, \ldots, y_N$ dans des listes `t` et `y` pour, éventuellement, tracer ensuite le graphique de la solution approchée :

```
def Euler(f,alpha,beta,u,N):
        y = [0]*(N+1)
        y[0] = u
        h = (beta - alpha)/N
        t = [alpha + k*h for k in range(N+1)]
        for k in range(N):
                y[k+1] = y[k] + h*f(t[k], y[k])
        return t,y
```

En pratique, la méthode d'Euler fonctionne plutôt bien pour les équations différentielle linéaire $y'=a(t)y+b(t)$ avec a,b de classe C^1 sur $]a,b[$, pour les équations différentielles de la forme $y'=f(y)$ avec f

bornée et de classe C^1 sur $\mathbb{R}$ et pour les équations $y' = f(t,y)$ avec f de classe C^1 sur $]a,b[\times\mathbb{R}$ telle qu'il existe $L>0$ vérifiant : $\forall x,y\in\mathbb{R}$, $|f(t,x)-f(t,y)|\leqslant L\times|x-y|$

On peut aussi appliquer la méthode d'Euler à certains systèmes différentiels comme celui ci-dessous (modèle proie-prédateur de Beddington) où $x(t)$ représente une population de proies et $y(t)$ celle de prédateurs au temps t, et où $m,r\in\mathbb{R}_+^*$. :

$$\begin{cases} x'(t) = rx(t) - 0.1\times\frac{x(t)y(t)}{1+0.1\times(x(t)+y(t))} \\ y'(t) = -my(t) + 0.1\times\frac{x(t)y(t)}{1+0.1\times(x(t)+y(t))} \end{cases}$$

Fonction Python prenant r, m, $x(0), y(0)$, T, $N\in\mathbb{N}^*$ en entrée, et donnant en sortie trois listes Temps, X et Y définies par :

- Temps[k] $= k\times\frac{T}{N}$ (pour $k\in[\![0,N]\!]$, Temps est la liste des temps),
- X[k] et Y[k] sont, pour $k\in[\![0,N]\!]$, les populations des proies et prédateurs à l'instant k.

```
1
2  def f(t,x,y,r,m):
3          return r*x-0.1*(x*y/(1+0.1*(x+y))), -m*y+0.1*(x*y/(1+0.1*(x+y)))
4
5  def Euler(r,m,x0,y0,T,N):
6          X, Y = [0]*(N+1), [0]*(N+1)
7          X[0], Y[0] = x0, y0
8          h = T/N
9          Temps = [k*h for k in range(N+1)]
10         for k in range(N):
11                 X[k+1] = X[k] + h*f(Temps[k], X[k], Y[k],r,m)[0]
12                 Y[k+1] = Y[k] + h*f(Temps[k], X[k], Y[k],r,m)[1]
13         return Temps,X,Y
```

Exercice 3

Soit (E) l'équation différentielle : $y'(t) = -5\sin(5t)\sqrt{1+y^2(t)}$. On admet que (E) admet une unique solution φ sur $\mathbb{R}$ telle que $\varphi(0) = \frac{e}{2} - \frac{1}{2e}$ et que cette solution est T-périodique où $0\leqslant T\leqslant 2$.

1) a) Écrire une fonction Python prenant en entrée un entier $N\geqslant 1$ et affichant le graphe d'une approximation de φ sur $[0,4]$ obtenu avec la méthode d'Euler avec un pas de $h=\frac{4}{N}$. Cette fonction devra également renvoyer en sortie :

- La liste $[t_0,t_1,\ldots,t_N]$ des abscisses,
- la liste $[\varphi(t_0),\varphi(t_1),\ldots,\varphi(t_N)]$ des ordonnées correspondantes.

b) Tester la fonction précédente avec $N=10$, $N=30$, $N=100$ et $N=1000$. À partir de quelle valeur de N le caractère T-périodique de φ avec $0\leqslant T\leqslant 2$ semble-t-il clair ?

2) Évaluer graphiquement T puis à l'aide d'une fonction Python pouvant faire appel à la fonction créée au 1) a).

Corrigés d'exercices d'analyse numérique élémentaire

Exercice 1

Pour $n \in \mathbb{N}^*$, on considère le polynôme $P_n = X^n - (1-X)^3$.

1) Montrer que, pour tout entier naturel $n \geqslant 1$, le polynôme P_n possède dans l'intervalle $]0,1[$ une unique racine que l'on notera α_n.

2) a) Écrire une fonction Python prenant en entrée n et donnant en sortie une valeur approchée de α_n à 10^{-6} près.

b) En déduire la liste des 20 premières valeurs de $(\alpha_n)_{n\geqslant 1}$. Que conjecturer ?

c) Déterminer la liste $[\alpha_{1000}, \alpha_{2000}, \alpha_{3000}, \ldots, \alpha_{10000}]$. Que conjecturer ?

Corrigé :

1) P_n est continue sur $[0,1]$ car dérivable sur $[0,1]$ et, pour $x \in]0,1[$, $P_n'(x) = nx^{n-1} + 3(1-x)^2 > 0$ car $x^{n-1} > 0$ et $1-x > 0$ donc $nx^{n-1} > 0$ et $3(1-x)^2 > 0$.

Ainsi, P_n réalise une bijection strictement croissante de $[0,1]$ sur $[P_n(0), P_n(1)] = [-1,1]$. Comme $0 \in]-1,1[$, il existe un unique réel $\alpha_n \in]0,1[$ tel que $P_n(\alpha_n) = 0$.

2) a) On peut utiliser une dichotomie.

```
def f(n,x):
    return x**n - (1-x)**3

def dichot(f,n):
    a, b = 0, 1
    while b-a > 10**(-6):
        c = (a+b)/2
        if f(n,a)*f(n,c) <= 0:
            b = c
        else:
            a = c
    return (a+b)/2
```

b) Avec le code :

```
L = [dichot(f,k) for k in range(1,21)]
print(L)
```

on obtient :
[0.3176722526550293, 0.4301600456237793, 0.4999995231628418, 0.5497002601623535, 0.587679386138916, 0.6180338859558105, 0.6430630683898926, 0.6641840934753418, 0.6823277473449707, 0.6981396675109863, 0.7120795249938965, 0.7244915962219238, 0.7356352806091309, 0.7457108497619629, 0.7548775672912598, 0.7632641792297363, 0.770972728729248, 0.7780900001525879, 0.7846856117248535, 0.7908205986022949]
La suite $(\alpha_n)_{n\geqslant 1}$ semble croissante.
c) Avec le code :

```
M = [dichot(f,k*1000) for k in range(1,11)]
print(M)
```

on obtient :
[0.9870457649230957, 0.9926562309265137, 0.9947619438171387, 0.9958882331848145, 0.9965958595275879, 0.9970850944519043, 0.9974446296691895, 0.9977211952209473, 0.9979405403137207, 0.9981188774108887]
La suite semble $(\alpha_n)_{n\geqslant 1}$ converger vers 1.

Exercice 2

Soit a, b deux réels tels que $a < b$ et $f : [a, b] \to \mathbb{R}$ une fonction continue et croissante. L'objectif de cet exercice est de démontrer que les suites de sommes de Riemann

$$\left(\frac{b-a}{n}\sum_{k=0}^{n-1} f\left(a+k\frac{b-a}{n}\right)\right)_{n\geqslant 1} \text{ et } \left(\frac{b-a}{n}\sum_{k=1}^{n} f\left(a+k\frac{b-a}{n}\right)\right)_{n\geqslant 1}$$

convergent bien vers $\int_a^b f(t)\mathrm{d}t$ et d'avoir une majoration de l'erreur commise par ces approximations.
Pour tout entier naturel n non nul, on pose

$$U_n = \frac{b-a}{n}\sum_{k=0}^{n-1} f\left(a+k\frac{b-a}{n}\right) \text{ et } V_n = \frac{b-a}{n}\sum_{k=1}^{n} f\left(a+k\frac{b-a}{n}\right).$$

1) Soit $n \in \mathbb{N}^*$.
a) Montrer : $\forall k \in [\![1, n]\!]$,

$$\frac{b-a}{n} f\left(a+(k-1)\frac{b-a}{n}\right) \leqslant \int_{a+(k-1)\frac{b-a}{n}}^{a+k\frac{b-a}{n}} f(x)\mathrm{d}x \leqslant \frac{b-a}{n} f\left(a+k\frac{b-a}{n}\right).$$

b) En déduire :

$$\forall n \in \mathbb{N}^*,\ U_n \leqslant \int_a^b f(x)\mathrm{d}x \leqslant V_n$$

2) Calculer et simplifier $V_n - U_n$. En déduire :

$$\forall n \in \mathbb{N}^*,\ 0 \leqslant \int_a^b f(x)\mathrm{d}x - U_n \leqslant \frac{b-a}{n}(f(b)-f(a)).$$

3) **Applications**

a) Écrire une fonction Python qui, pour une fonction f continue et croissante sur un intervalle $[a,b]$, donne en sortie une approximation de $\int_a^b f(t)\mathrm{d}t$ à ε-près où f, a, b, ε seront fournis en entrée de la fonction (avec $\varepsilon \in \mathbb{R}_+^*$).
b) En déduire une approximation à 10^{-5} près des intégrales $\int_2^6 \frac{e^t}{t^2}\mathrm{d}t$, $\int_0^2 e^x \arctan(x)\mathrm{d}x$ et $\int_4^{10} \sqrt{1+\arctan(x)\ln(x+2)}\mathrm{d}x$.

Corrigé :
1) Soit $k \in [\![1,n]\!]$. Pour $x \in \left[a+(k-1)\frac{b-a}{n}, a+k\frac{b-a}{n}\right]$, on a par croissance de f sur $\left[a+(k-1)\frac{b-a}{n}, a+k\frac{b-a}{n}\right]$

$$f\left(a+(k-1)\frac{b-a}{n}\right) \leqslant f(x) \leqslant f\left(a+k\frac{b-a}{n}\right).$$

Comme $a+(k-1)\frac{b-a}{n} \leqslant a+k\frac{b-a}{n}$, on a par croissance de l'intégrale

$$\int_{a+(k-1)\frac{b-a}{n}}^{a+k\frac{b-a}{n}} f\left(a+(k-1)\frac{b-a}{n}\right)dx \leqslant \int_{a+(k-1)\frac{b-a}{n}}^{a+k\frac{b-a}{n}} f(x)dx \leqslant \int_{a+(k-1)\frac{b-a}{n}}^{a+k\frac{b-a}{n}} f\left(a+k\frac{b-a}{n}\right)dx.$$

Or, pour une constante réelle c, $\int_{a+(k-1)\frac{b-a}{n}}^{a+k\frac{b-a}{n}} c dx = [cx]_{a+(k-1)\frac{b-a}{n}}^{a+k\frac{b-a}{n}} = c \times \frac{b-a}{n}$ donc

$$\frac{b-a}{n} f\left(a+(k-1)\frac{b-a}{n}\right) \leqslant \int_{a+(k-1)\frac{b-a}{n}}^{a+k\frac{b-a}{n}} f(x)dx \leqslant \frac{b-a}{n} f\left(a+k\frac{b-a}{n}\right).$$

b) En sommant les encadrements précédents pour $k \in [\![1,n]\!]$,

$$\frac{b-a}{n}\sum_{k=1}^{n} f\left(a+(k-1)\frac{b-a}{n}\right) \leqslant \sum_{k=1}^{n}\int_{a+(k-1)\frac{b-a}{n}}^{a+k\frac{b-a}{n}} f(x)dx \leqslant \frac{b-a}{n}\sum_{k=1}^{n} f\left(a+k\frac{b-a}{n}\right)$$

or, par changement d'indice $j=k-1$, on a $\frac{b-a}{n}\sum_{k=1}^{n} f\left(a+(k-1)\frac{b-a}{n}\right) = \frac{b-a}{n}\sum_{j=0}^{n-1} f\left(a+j\frac{b-a}{n}\right) = U_n$ et, d'après la relation de Chasles,

$$\sum_{k=1}^{n}\int_{a+(k-1)\frac{b-a}{n}}^{a+k\frac{b-a}{n}} f(x)dx = \int_a^b f(x)dx$$

donc on a bien

$$\forall n \in \mathbb{N}^*,\ U_n \leqslant \int_a^b f(x)\mathrm{d}x \leqslant V_n.$$

2) Soit $n \in \mathbb{N}^*$. On a

$$\begin{aligned} V_n - U_n &= \frac{b-a}{n}\left(\sum_{k=1}^{n} f\left(a+k\frac{b-a}{n}\right) - \sum_{k=0}^{n-1} f\left(a+k\frac{b-a}{n}\right)\right) \\ &= \frac{b-a}{n}\left(f\left(a+n\frac{b-a}{n}\right) - f\left(a+0\times\frac{b-a}{n}\right)\right) \\ &= \frac{b-a}{n}(f(b)-f(a)). \end{aligned}$$

En retranchant U_n à chaque membre de l'encadrement $U_n \leqslant \int_a^b f(x)\mathrm{d}x \leqslant V_n$, on obtient $0 \leqslant \int_a^b f(x)\mathrm{d}x - U_n \leqslant V_n - U_n$ donc

$$0 \leqslant \int_a^b f(x)\mathrm{d}x - U_n \leqslant \frac{b-a}{n}(f(b)-f(a)).$$

3) a) D'après le dernier encadrement, U_n est une valeur approchée de $\int_a^b f(x)\mathrm{d}x$ à ε près dès que $\frac{b-a}{n}(f(b)-f(a)) \leqslant \varepsilon$.

```python
def Approx(f,a,b,eps):
    # Calcul du n adéquat
    n = 1
    while (b-a)/n*(f(b)-f(a)) > eps:
        n += 1
    # Calcul de Un:
    Pas = (b-a)/n
    Somme = 0
    for k in range(n):
        Somme += f(a+k*Pas)
    return Pas*Somme
```

b) Après avoir vérifié que les fonctions intégrées sont effectivement croissantes (et continues) sur leurs intervalles d'intégration, le code

```python
from math import *

def f1(x):
    return exp(x)/x**2

def f2(x):
    return exp(x)*atan(x)

def f3(x):
    return sqrt(1+atan(x)*log(x+2))

eps = 10**(-5)
print(Approx(f1,2,6,eps))
print(Approx(f2,0,2,eps))
print(Approx(f3,4,10,eps))
```

fournit :
$\int_2^6 \frac{e^t}{t^2}\mathrm{d}t \simeq 17.49192$, $\int_0^2 e^x \arctan(x)\mathrm{d}x \simeq 5.51759$ et $\int_4^{10} \sqrt{1+\arctan(x)\ln(x+2)}\mathrm{d}x \simeq 12.13626$.

Exercice 3

Soit (E) l'équation différentielle : $y'(t) = -5\sin(5t)\sqrt{1+y^2(t)}$. On admet que (E) admet une unique solution φ sur $\mathbb{R}$ telle que $\varphi(0) = \frac{e}{2} - \frac{1}{2e}$ et que cette solution est T-périodique où $0 \leqslant T \leqslant 2$.
1) a) Écrire une fonction Python prenant en entrée un entier $N \geqslant 1$ et affichant le graphe d'une approximation de φ sur $[0,4]$ obtenu avec la méthode d'Euler avec un pas de $h = \frac{4}{N}$. Cette fonction devra également renvoyer en sortie :

- La liste $[t_0, t_1, \ldots, t_N]$ des abscisses,
- la liste $[\varphi(t_0), \varphi(t_1), \ldots, \varphi(t_N)]$ des ordonnées correspondantes.

b) Tester la fonction précédente avec $N = 10$, $N = 30$, $N = 100$ et $N = 1000$. À partir de quelle valeur de N le caractère T-périodique de φ avec $0 \leqslant T \leqslant 2$ semble-t-il clair ?
2) Évaluer graphiquement T puis à l'aide d'une fonction Python pouvant faire appel à la fonction créée au 1) a).

Corrigé :
1) a)

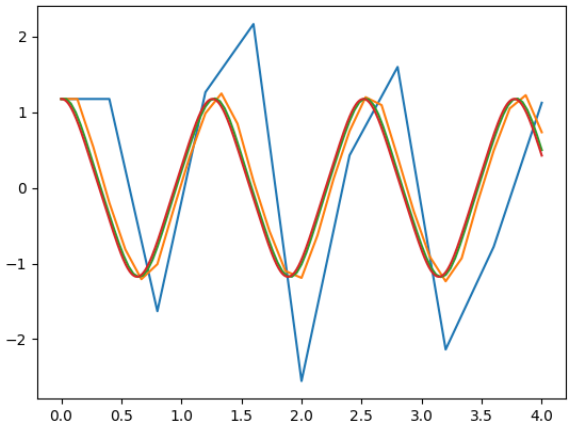
```
from math import *
import matplotlib.pyplot as plt

def Euler(N):
    def f(x,y):
        return -5*sin(5*x)*sqrt(1+y**2)
    y = [0]*(N+1)
    y[0] = e/2-1/(2*e)
    h = 4/N
    t = [k*h for k in range(N+1)]
    for k in range(N):
        y[k+1] = y[k] + h*f(t[k], y[k])
    plt.plot(t,y)
    plt.show()
    return t,y
```

b)

```
for N in [10,30,100,1000]:
    Euler(N)
```

À partir de $N = 100$, la courbe est tout à fait lisse et le caractère périodique de la solution semble

manifeste.



2) Graphiquement, en pointant les deux premiers points où φ atteint son minimum, on obtient $T \simeq 1.88 - 0.625 = 1.255$.
Pour avoir plus de précision, on peut aussi utiliser le code suivant qui détermine (à partir des listes t et y créées par la fonction du 1) a)) à quels premières abscisses t_1 et t_2, la fonction φ atteint son minimum (t_1 est le premier $t[k]$ tel que $y[k+1] > y[k]$ et $y[k] < y[k-1]$ et t_2 est le deuxième $t[k]$ vérifiant cette même propriété).

```
def Periode():
    N = 10**6
    t,y = Euler(N)
    t1Defini = False
    for k in range(N):
        if y[k]<y[k+1] and y[k]<y[k-1] and not(t1Defini):
            t1 = t[k]
            t1Defini = True
        elif y[k]<y[k+1] and y[k]<y[k-1] and t1Defini:
            t2 = t[k]
            print(t1,t2)
            return t2-t1

# Appel de la fonction
Periode()
```

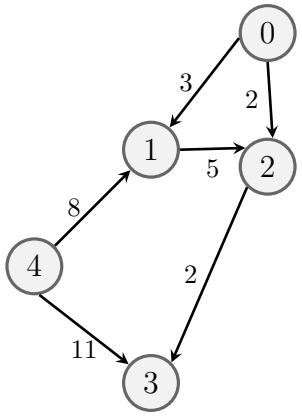
On a utilisé $N = 10^6$ pour avoir une bonne précision, ce qui fournit $t_1 \simeq 0.62832$ et $t_2 \simeq 1.884956$ donc $T \simeq t_2 - t_1 = 1.256636$.

L'algorithme de Dijkstra

Cours sur l'algorithme de Dijkstra

Algorithme de Dijkstra :

On considère un graphe orienté pondéré comme, par exemple, celui ci-dessous :



Un chemin d'un noeud à un autre est une succession d'arête permettant de relier ces deux noeuds (en respectant l'orientation des arêtes). La longueur d'un chemin est la somme des poids des arêtes constituant le chemin.
Étant donné un noeud d'origine fixé (comme le noeud 0 dans notre exemple), l'algorithme de Dijkstra a pour objectif de déterminer les plus courts chemins permettant de relier le noeud d'origine à n'importe quel autre noeud du graphe.
Pour comprendre le fonctionnement de l'algorithme de Dijkstra, il faut imaginer les arêtes comme des traînées de poudre prêtes à brûler :
on met le feu au noeud d'origine et le feu se propage à vitesse constante ; si le feu atteint un noeud donné, il se propage en même temps aux arêtes partant de ce noeud. Avec ce modèle, le poids de chaque arête correspondrait au temps mis par le feu pour parcourir l'arête (dans l'orientation indiquée par l'arête).
Le principe de l'algorithme de Dijkstra consiste à répertorier les noeuds en train de brûler et les prochains à l'être (en prenant en compte les instants où un noeud brûle ou pourrait brûler) : dès qu'un noeud A brûle à un instant t, le chemin des flammes qui a mené jusqu'à ce noeud à partir du noeud d'origine correspond à un plus court chemin jusqu'à A.
Pour pouvoir appliquer l'algorithme de Dijkstra il faudra supposer (conformément à notre modèle) que les poids de chaque arêtes sont positifs.

Exercice 1 : **Saisie du graphe**

Écrire une fonction Python prenant en entrée deux entiers naturels N et n, ainsi qu'une liste L. L'entier naturel N est le nombre de noeuds du graphe orienté pondéré et n est le nombre d'arêtes de ce même graphe. La liste L devra être formée de triplets $(n1, n2, p)$ où $n1$ et $n2$ sont deux noeuds reliés par une arête (orientée de $n1$ à $n2$) et p est le poids de cette arête (la liste L répertorie donc l'ensemble des données des arêtes du graphe).
La fonction devra donner en sortie la représentation `ListAdj` du graphe sous forme de liste d'adjacence. On rappelle que `ListAdj[k]` doit être la liste des couples (n, p) tels qu'il existe une arête de k à n de poids p.

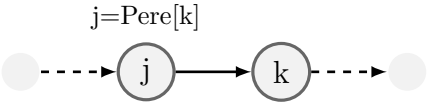
Exemple :
En entrée :
`N = 4, n = 6`

```
L = [(0,2,3),(1,2,2),(2,0,5),(1,3,1),(3,2,7),(0,3,11)]
```
En sortie :
```
ListeAdj = [[(2,3),(3,11)],[(2,2),(3,1)],[(0,5)],[(2,7)]]
```

On va écrire dans l'exercice suivant l'initialisation des variables utiles.
Nous aurons besoin de 6 variables : `Graphe, nbNoeuds, distances, Peres, noeudOrigine, distNoeuds`.

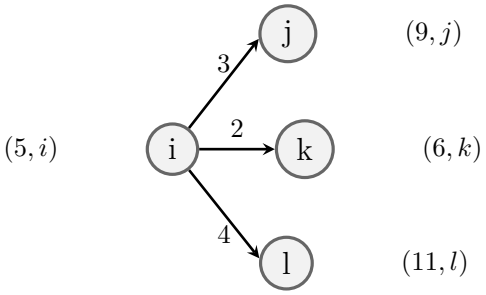
- `Graphe` sera le graphe sous forme de liste d'adjacence.
- `nbNoeuds` sera le nombre de noeuds (donc la taille de la liste d'adjacence).
- `distance` est la liste des longueurs des plus courts chemins à partir du noeud d'origine (éventuellement recalculées).

  Ainsi, cette liste a autant d'éléments que de noeuds dans le graphe et `distance[k]` est la distance du noeud d'origine au noeud `k`.
- `Peres` est la liste des antécédents des noeuds dans un plus court chemin. Plus précisément, `Peres` a autant d'éléments qu'il y a de noeuds dans le graphe, et `Peres[k]` est le numéro u d'un noeud d'une arête de u à `k` dans un plus court chemin (les éléments de `Peres` sont éventuellement recalculés au cours de l'algorithme).

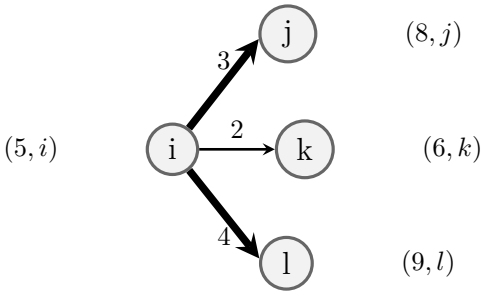
  

  Il faut bien noter que `Peres[k]` définira donc une arête d'un plus court chemin constitué de cette arête (cf schéma ci-dessus), peu importe le noeud de fin du chemin. Cela est licite car il n'est pas difficile de montrer qu'un plus court chemin d'un noeud i à un noeud j passant par k est constitué d'un sous-chemin qui est un plus court chemin de i à k.
- `noeudOrigine` sera le noeud d'origine (demandé à l'utilisateur).
- `distNoeuds` sera une structure de donnée particulière (appelée file à priorité). Pour faire simple, ce sera une liste constituée de couples (d, x) où x est le numéro d'un noeud et d est la longueur d'un plus court chemin du noeud d'origine à x au moment où ce couple a été inséré dans `distNoeuds` (il sera en effet possible d'avoir au moins deux couples (d, x) et (d', x) dans `distNoeuds` avec $d \neq d'$). De plus, le premier élément extrait de `distNoeuds` sera toujours un couple (d, x) avec d minimal parmi tous les couples possibles (c'est ce qui fait qu'on appelle cette liste une file à priorité).

Le principe de l'algorithme de Dijkstra va consister à affecter à chaque noeud A une distance qui va correspondre à la longueur d'un plus court chemin provisoire du noeud d'origine à ce noeud A. Cette distance évolue au cours de l'algorithme et est éventuellement recalculée au cours de celui-ci (via la liste `distance` et la file à priorité `distNoeuds`).
Expliquons comment ses distances sont recalculées sur un exemple. On suppose ici que le noeud i est relié à trois autres noeuds j, k, l par des arêtes. On suppose qu'un plus court chemin jusqu'à i est de longueur 5 (avec notre image de la poudre : cela signifie que le noeud i sera brûlé à l'instant 5) et que les plus courts chemins calculés jusqu'ici pour j, k et l sont de longueurs respectives $9, 6$ et 11 (cela signifie que les noeuds j, k et l brûleraient au plus tard aux instants 9, 6 et 11).



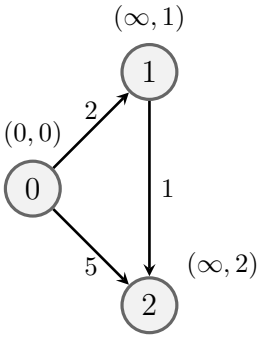
On effectue alors une opération dite de *relâchement* sur les arêtes si un des noeuds peut être brûlé plus tôt.



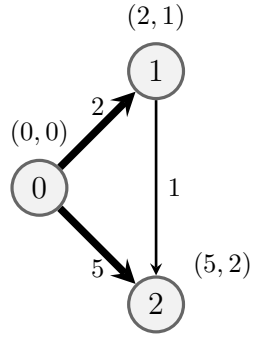
Relâchement de deux arêtes

On voit ici que i peut être brûlé à l'instant $5 + 3 = 8$ et comme 8 est inférieur à 9, on met à jour le couple $(9, j)$ qui devient $(8, j)$ (et `Peres[j]` devient i). De même, le noeud l peut être brûlé à l'instant $5 + 4 = 9$ et comme $9 < 11$, on met à jour le couple $(11, l)$ qui devient $(9, l)$ (et `Peres[l]` devient i). Par contre, le noeud k ne peut pas brûler plus tôt à partir de i : en effet, $5 + 2 = 7$ et $7 > 6$ donc on ne met pas à jour le couple $(6, k)$ (de même, on ne met pas à jour `Peres[k]`).

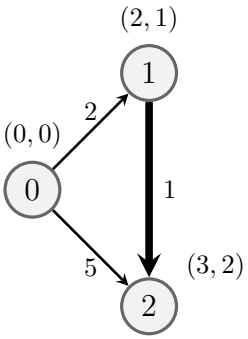
Voyons maintenant une illustration de l'algorithme sur un exemple complet mais tout simple, avec trois noeuds (les arêtes relâchées sont toujours en gras) :



`distances = [0,`$+\infty, +\infty$`]`, `distNoeuds=[(0,0)]`, `NoeudCourant=0`



`distances = [0,2,5]`, `distNoeuds=[(2,1),(5,2)]`, `NoeudCourant=1`



`distances = [0,2,3], distNoeuds=[(3,2),(5,2)], NoeudCourant=2`
`distances = [0,2,3], distNoeuds=[(5,2)], NoeudCourant=2`
`distances = [0,2,3], distNoeuds=[], NoeudCourant=2`

Exercice 2 : **Initialisation des variables**

Écrire une fonction `Initialisation` sans variable d'entrée ayant pour objectif d'initialiser les 6 variables `Graphe, nbNoeuds, distances, Peres, noeudOrigine` et `distNoeuds`.

- La variable `Graphe` sera définie comme la valeur de retour de la fonction de l'exercice 1.
- La variable `nbNoeuds` sera définie comme le nombre de noeuds du graphe (c'est la taille de la liste `Graphe`).
- Tous les éléments de `Peres` seront définis comme étant égaux à -1 (au départ, aucun plus court chemin n'a été défini).
- La valeur de `noeudOrigine` sera demandée à l'utilisateur (attention, ce doit être un entier).
- Les éléments de la liste `distances` seront initialisés à $+\infty$ (tous les noeuds sont considérés comme étant au départ à une distance infinie du noeud d'origine) sauf celui d'indice `noeudOrigine` qui sera initialisé à 0 (le noeud `noeudOrigine` est à une distance nulle de lui-même).
- `distNoeuds` est une structure de donnée particulière (on a dit que c'était une file à priorité).
  Voici un exemple d'implémentation d'une telle structure de donnée (nommée ici `h`). On a besoin d'importer la bibliothèque `heapq`.

```
1
2  from heapq import *
3
4  h = [] # initialisation de la file à priorité h
5  heappush(h, (3,5)) # insère (3,5) dans h
6  heappush(h, (2,7)) # insère (2,7) dans h
7  print(len(h)) # affiche la taille de h (ici 2)
8  print(heappop(h)) # affiche le minimum de h et le retire (ici, c'est (2,7))
9  print(len(h)) # affiche 1
10 print(heappop(h)) # affiche le minimum de h et le retire (ici, c'est (3,5))
11 print(len(h)) # affiche 0: h est vide.
```

`distNoeuds` ne devra contenir au départ que le couple (0, `noeudOrigine`).

La fonction `Initialisation` devra donner en sortie les 6 variables initialisées.

Exercice 3 : **Relâchement des arêtes**
Écrire une fonction Python `Relachement` qui prend en entrée x et d où x est le numéro d'un noeud et d est un réel positif (longueur d'un plus court chemin provisoire jusqu'à x).
Cette fonction déclarera les variables `Graphe, distances, Peres` et `distNoeuds` comme des variables globales.

```
1
2 def Relachement(x,d):
3     global Graphe, distances, Peres, distNoeuds
4     # ...
```

Cette fonction doit relâcher toutes les arêtes partant du noeud x. Concrètement, si y est l'extrémité d'une arête relachée partant de x, on redéfinira `distances[y]`, `Peres[y]` et on placera le couple `(distances[y],y)` dans la file à priorité `distNoeuds`.

Exercice 4 : **Synthèse**
En appelant au besoin les fonctions précédentes, traduire alors en Python la fonction `Dijkstra` suivante (sans variable d'entrée).

```
Fonction Dijkstra()
  Initialisation de Graphe, nbNoeuds, distances, Peres, noeudOrigine et distNoeuds
  Tant que la file à priorité distNoeuds n'est pas vide
    Retirer le premier couple (d,x) de distNoeuds (x est alors le noeud courant)
    Si d = distances[x]
      relâcher les arêtes partant de x (et mettre à jour Peres et distNoeuds)
```

Cette fonction déclarera les variables `Graphe, distances, Peres` et `distNoeuds` comme des variables globales.
Elle donnera en sortie `noeudOrigine, Peres` et `distances`.

Exercice 5 : **Afficher les plus courts chemins à la demande**
Écrire une fonction `AfficheChemPC(x)` prenant en entrée un noeud `x` et donnant en sortie :
la longueur d'un plus court chemin du noeud d'origine à `x`.
la donnée d'un tel chemin sous forme d'une liste de noeuds (depuis le noeud d'origine).
Cette fonction pourra faire appel à `Dijkstra()`.

Corrigés d'exercices sur l'algorithme de Dijkstra

Algorithme de Dijkstra :

Exercice 1 : **Saisie du graphe**
Écrire une fonction Python prenant en entrée deux entiers naturels N et n, ainsi qu'une liste L. L'entier naturel N est le nombre de noeuds du graphe orienté pondéré et n est le nombre d'arêtes de ce même

graphe. La liste L devra être formée de triplets $(n1, n2, p)$ où $n1$ et $n2$ sont deux noeuds reliés par une arête (orientée de $n1$ à $n2$) et p est le poids de cette arête (la liste L répertorie donc l'ensemble des données des arêtes du graphe). La fonction devra donner en sortie la représentation `ListAdj` du graphe sous forme de liste d'adjacence. On rappelle que `ListAdj[k]` doit être la liste des couples (n, p) tels qu'il existe une arête de k à n de poids p.

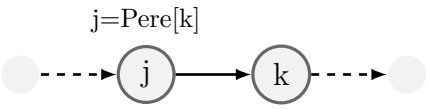
Exemple :
En entrée :
`N = 4, n = 6, L = [(0,2,3),(1,2,2),(2,0,5),(1,3,1),(3,2,7),(0,3,11)]`
En sortie :
`ListeAdj = [[(2,3),(3,11)],[(2,2),(3,1)],[(0,5)],[(2,7)]]`

Corrigé :

```
1
2  ## Exercice 1
3
4  def Saisie(N,n,L):
5      ListeAdj = [[] for k in range(N)]
6      for k in range(n):
7          n1, n2, p = L[k]
8          ListeAdj[n1].append((n2,p))
9      return ListeAdj
```

On va écrire dans l'exercice suivant l'initialisation des variables utiles.
Nous aurons besoin de 6 variables : `Graphe, nbNoeuds, distances, Peres, noeudOrigine, distNoeuds`.

- `Graphe` sera le graphe sous forme de liste d'adjacence.
- `nbNoeuds` sera le nombre de noeuds (donc la taille de la liste d'adjacence).
- `distance` est la liste des longueurs des plus courts chemins à partir du noeud d'origine (éventuellement recalculées). Ainsi, cette liste a autant d'éléments que de noeuds dans le graphe et `distance[k]` est la distance du noeud d'origine au noeud `k`.
- `Peres` est la liste des antécédents des noeuds dans un plus court chemin. Plus précisément, `Peres` a autant d'éléments qu'il y a de noeuds dans le graphe, et `Peres[k]` est le numéro u d'un noeud d'une arête de u à `k` dans un plus court chemin (les éléments de `Peres` sont éventuellement recalculés au cours de l'algorithme).



Il faut bien noter que `Peres[k]` définira donc une arête d'un plus court chemin constitué de cette arête (cf schéma ci-dessus), peu importe le noeud de fin du chemin. Cela est licite car il n'est pas difficile de montrer qu'un plus court chemin d'un noeud i à un noeud j passant par k est constitué d'un sous-chemin qui est un plus court chemin de i à k.

- `noeudOrigine` sera le noeud d'origine (demandé à l'utilisateur).
- `distNoeuds` sera une structure de donnée particulière (appelée file à priorité). Pour faire simple, ce sera une liste constituée de couples (d, x) où x est le numéro d'un noeud et d est la longueur d'un plus court chemin du noeud d'origine à x au moment où ce couple a été inséré dans `distNoeuds` (il sera en effet possible d'avoir au moins deux couples (d, x) et (d', x) dans `distNoeuds` avec $d \neq d'$). De plus, le premier élément extrait de `distNoeuds` sera toujours un couple (d, x) avec d minimal parmi tous les couples possibles (c'est ce qui fait qu'on appelle cette liste une file à priorité).

Exercice 2 : **Initialisation des variables**

Écrire une fonction `Initialisation` sans variable d'entrée ayant pour objectif d'initialiser les 6 variables `Graphe, nbNoeuds, distances, Peres, noeudOrigine` et `distNoeuds`.

- La variable `Graphe` sera définie comme la valeur de retour de la fonction de l'exercice 1.
- La variable `nbNoeuds` sera définie comme le nombre de noeuds du graphe (c'est la taille de la liste `Graphe`).
- Tous les éléments de `Peres` seront définis comme étant égaux à -1 (au départ, aucun plus court chemin n'a été défini).
- La valeur de `noeudOrigine` sera demandée à l'utilisateur (attention, ce doit être un entier).
- Les éléments de la liste `distances` seront initialisés à $+\infty$ (tous les noeuds sont considérés comme étant au départ à une distance infinie du noeud d'origine) sauf celui d'indice `noeudOrigine` qui sera initialisé à 0 (le noeud `noeudOrigine` est à une distance nulle de lui-même).
- `distNoeuds` est une structure de donnée particulière (on a dit que c'était une file à priorité).

  Voici un exemple d'implémentation d'une telle structure de donnée (nommée ici `h`). On a besoin d'importer la bibliothèque `heapq`.

```
1
2  from heapq import *
3
4  h = [] # initialisation de la file à priorité h
5  heappush(h, (3,5)) # insère (3,5) dans h
6  heappush(h, (2,7)) # insère (2,7) dans h
7  print(len(h)) # affiche la taille de h (ici 2)
8  print(heappop(h)) # affiche le minimum de h et le retire (ici, c'est (2,7))
9  print(len(h)) # affiche 1
10 print(heappop(h)) # affiche le minimum de h et le retire (ici, c'est (3,5))
11 print(len(h)) # affiche 0: h est vide.
```

  `distNoeuds` ne devra contenir au départ que le couple (0,`noeudOrigine`).

La fonction `Initialisation` devra donner en sortie les 6 variables initialisées.

Corrigé :

```
1
2  ## Exercice 2
3  from heapq import *
4
5  def Initialisation():
6      nbNoeuds = int(input("nombre de noeuds?"))
7      n = int(input("nombre d'arêtes?"))
8      L = eval(input("Liste des arêtes?"))
9      Graphe = Saisie(nbNoeuds,n,L)
10     Peres = [-1]*nbNoeuds
11     noeudOrigine = int(input("noeud d'origine?"))
12     distances = [float('inf')]*nbNoeuds
13     distances[noeudOrigine] = 0
14     distNoeuds = []
15     heappush(distNoeuds, (0,noeudOrigine))
16     return Graphe, nbNoeuds, distances, Peres, noeudOrigine, distNoeuds
```

Exercice 3 : **Relâchement des arêtes**

Écrire une fonction Python `Relachement` qui prend en entrée x et d où x est le numéro d'un noeud et d est un réel positif (longueur d'un plus court chemin provisoire jusqu'à x). Cette fonction déclarera les variables `Graphe, distances, Peres` et `distNoeuds` comme des variables globales.

```
1
2  def Relachement(x,d):
3      global Graphe, distances, Peres, distNoeuds
4      # ...
```

Cette fonction doit relâcher toutes les arêtes partant du noeud x. Concrètement, si y est l'extrémité d'une arête relachée partant de x, on redéfinira `distances[y]`, `Peres[y]` et on placera le couple `(distances[y],y)` dans la file à priorité `distNoeuds`.

Corrigé :

```
1
2  ## Exercice 3
3  def Relache(x,d):
4      global Graphe, distances, Peres, distNoeuds
5      for y, poids in Graphe[x]:
6          if d + poids < distances[y]:
7              distances[y] = d + poids
8              Peres[y] = x
9              heappush(distNoeuds, (distances[y],y))
```

Exercice 4 : **Synthèse**

En appelant au besoin les fonctions précédentes, traduire alors en Python la fonction `Dijkstra` suivante (sans variable d'entrée).

```
Fonction Dijkstra()
  Initialisation de Graphe, nbNoeuds, distances, Peres, noeudOrigine et distNoeuds
  Tant que la file à priorité distNoeuds n'est pas vide
    Retirer le premier couple (d,x) de distNoeuds (x est alors le noeud courant)
    Si d = distances[x]
      relâcher les arêtes partant de x (et mettre à jour Peres et distNoeuds)
```

Cette fonction déclarera les variables `Graphe, distances, Peres` et `distNoeuds` comme des variables globales.
Elle donnera en sortie `noeudOrigine, Peres` et `distances`.

Corrigé :

```python

## Exercice 4
def Dijkstra():
    global Graphe, distances, Peres, distNoeuds
    Graphe, nbNoeuds, distances, Peres, noeudOrigine, distNoeuds = Initialisation()
    while len(distNoeuds) > 0:
        d, x = heappop(distNoeuds)
        if d == distances[x]:
            Relache(x,d)
    return noeudOrigine, Peres, distances
```

Exercice 5 : **Afficher les plus courts chemins à la demande**

Écrire une fonction `AfficheChemPC(x)` prenant en entrée un noeud `x` et donnant en sortie :
la longueur d'un plus court chemin du noeud d'origine à `x`.
la donnée d'un tel chemin sous forme d'une liste de noeuds (depuis le noeud d'origine).
Cette fonction pourra faire appel à `Dijkstra()`.

Corrigé :

```python

## Exercice 5
def AfficheChemPC(x):
    noeudOrigine, Peres, distances =  Dijkstra()
    noeud = x
    ChemAlenvers = [noeud]
    while Peres[noeud] != -1:
        noeud = Peres[noeud]
        ChemAlenvers.append(noeud)
    return distances[x], [ChemAlenvers[k] for k in range(len(ChemAlenvers)-1,-1,-1)]
```