AGR. Algorithmes. Bases

I/ Parcours séquentiel d'une liste

1/ Calcul d'une moyenne

Parcourir une liste **séquentiellement**, c'est la parcourir élément par élément, dans leur ordre d'apparition.

Méthode : Pour calculer une **moyenne**, on additionne tous les éléments de la liste et on divise par le nombre de ses termes.

Moyenne d'une liste de nombres générés aléatoirement :

```
from random import randint

# Génération d'une liste de 20 éléments aléatoires
liste = [randint(0,20) for i in range(20)]

def moyenne(liste) :
    n = len(liste)   # Nombre d'éléments de la liste
    somme = 0        # Récupère la somme de tous les termes (0 au départ)
    for nombre in liste :
        somme = somme + nombre # Addition successive de chaque terme

    return somme/n

print("Liste :",liste)
print("Moyenne :", moyenne(liste))
```

```
Liste : [13, 13, 9, 5, 11, 11, 8, 12, 7, 11, 0, 8, 2, 4, 11, 16, 8, 17, 8, 4]
Moyenne : 8.9
```

Remarque : le principe est le même pour les listes de listes, il y aura alors deux boucles for imbriquées :

```
from random import randint

# Génération d'une liste de 10 listes de 5 éléments chacune
listes = [[randint(0,20) for i in range(5)] for j in range(10)]

def moyenne(listes) :
    somme = 0        # Récupère la somme de tous les termes (0 au départ)
    compteur = 0     # Compte le nombre d'éléments de listes
    for liste in listes : # Pour chaque liste de liste,
          for nombre in liste :  # on parcourt chaque élément
                compteur = compteur + 1  # Ajoute 1 au compteur d'éléments
                somme = somme + nombre   # Addition successive de chaque terme

    return somme/compteur

print("Liste :",listes)
print("Moyenne :", moyenne(listes))
```

```
Liste : [[5, 10, 11, 6, 12], [10, 1, 11, 13, 13], [12, 14, 19, 7, 11], [10, 1, 17, 7, 1], [16, 15
9], [1, 10, 9, 20, 1], [10, 14, 2, 5, 1], [10, 18, 7, 11, 8], [19, 3, 3, 18, 9]]
Moyenne : 9.52
```

2/ Recherche d'une occurrence

Il s'agit de rechercher si une valeur est bien dans une liste ou non. Il peut s'agir de nombres mais aussi de chaînes de caractères.

Un exemple de programme de recherche de prénoms ici :

```
# Génération d'une liste de prénoms
liste = ["Chloé","Tristan","Ben","Louise","Marie","Henri"]

def recherche(liste,valeur) :
    for prenom in liste :
        if prenom == valeur :
            return True
    return False

print(recherche(liste,"Ben")) # Affiche True
print(recherche(liste,"Anaïs")) # Affiche False
```

```
True
False
```

Le langage Python offre deux possibilités toutes faites :

- L'expression ***valeur in list*** qui renvoie *True* si la valeur est dans la liste et *False* le cas échéant.
- La méthode ***count(valeur)*** qui renvoie le nombre de fois où la valeur est présente dans la liste.

Un exemple ici :

```
from random import randint

# Génération d'une liste de nombres aléatoires.
liste = [randint(0,20) for i in range(25)]

def recherche(liste,valeur) :  # Vérifie la présence d'une valeur.
    return valeur in liste


def nombreOccurences(liste, valeur) : # Renvoie le nombre d'occurences
    return liste.count(valeur)       # de la valeur dans la liste.

print("Liste :",liste)
print(recherche(liste,3))               # Affiche true
print(nombreOccurences(liste,3),"fois") # Affiche 3 fois
print(recherche(liste,30))              # Affiche false
print(nombreOccurences(liste,30))       # Affiche 0
```

```
Liste : [20, 18, 19, 7, 17, 10, 5, 13, 3, 8, 14, 0, 15, 8, 0, 16, 3, 10, 3, 4, 15, 16, 18, 17, 6]
True
3 fois
False
0
```

3/ Recherche d'un extremum

Un **extremum** est la valeur minimale ou maximale d'une liste.

Voici un programme de recherche d'un maximum d'une liste de nombres :

```
from random import randint

# Génération d'une liste de nombres aléatoires.
liste = [randint(0,20) for i in range(25)]

def maximum(liste) :
    plusGrand = liste[0]  # On prend par défaut le premier élément
    for nombre in liste :
        if nombre > plusGrand :
            plusGrand = nombre
    return plusGrand

print("Liste :",liste)
print("Maximum :",maximum(liste))
```

```
Liste : [9, 4, 6, 5, 19, 3, 17, 15, 3, 5, 10, 3, 7, 3, 0, 14, 17, 6, 1, 20, 6, 8, 20, 12, 19]
Maximum : 20
```

Remarque : la programmation est similaire pour trouver le minimum d'une liste, il suffit de changer le « < » en « > ».

Le langage Python fournit une fonction qui permet de trouver les extremum d'une liste :

```
from random import randint

# Génération d'une liste de nombres aléatoires.
liste = [randint(0,20) for i in range(25)]

def minimum(liste) :
    return min(liste)

def maximum(liste) :
    return max(liste)

print("Liste :",liste)
print("Minimum :", minimum(liste))
print("Maximum :", maximum(liste))
```

```
Liste : [6, 11, 14, 16, 4, 4, 10, 0, 6, 3, 7, 20, 6, 13, 14, 2, 2, 12, 16, 8, 10, 20, 13, 6, 2]
Minimum : 0
Maximum : 20
```

Remarque : on utilise le même type d'algorithme pour déterminer une valeur approchée d'une fonction *f* définie sur un intervalle [a , b[avec a , b deux nombres réels et a < b.

A savoir : attention, il s'agit bien de l'intervalle [a , b[et non de [a , b]. Dans ce cas, *f*(b) ne sera pas traité !

Voici un exemple de programme avec une fonction *f* définie par $f(x) = 3x^2 + 1x - 7$. On cherche le minimum sur l'intervalle [-1 , 2[avec un pas de 0,001.

```
def f(x) :
    return 3*x**2 + x - 7

def minimum(borneMin,borneMax,pas) :
    absCourante = borneMin # Première valeur des abscisses.
    minOrd = f(absCourante)  # Première valeur testée en tant que minimum par défaut.

    while absCourante < borneMax : # On respecte les bornes de l'intervalle.
        absCourante = absCourante + pas  # On teste la valeur suivante
        if f(absCourante) < minOrd :  # Si on trouve une image inférieure à minOrd,
            minOrd = f(absCourante)   # on a un nouveau minimum.

    return minOrd

borneMin = -1  # Borne minimale.
borneMax = 2   # Borne maximale.
pas = 0.001    # On teste les valeurs de l'intervalle de 0.001 en 0.001.

print("Minimum :",minimum(borneMin,borneMax,pas))
```

```
Minimum : -7.083333
```

Remarques :

- Il s'agit de valeurs approchées, d'autant plus précises que le pas est faible.
- Le programme peut être amélioré et renvoyer aussi la valeur de l'intervalle [a , b[pour laquelle la fonction *f* admet un minimum : c'est l'objet d'un exercice.
- Comme précédemment, la recherche d'un maximum d'une fonction sur un intervalle suit le même type d'algorithme.

II/ Recherche dichotomique

La recherche dichotomique s'effectue sur un tableau **trié**. En langage Python, le plus simple est d'utiliser l'objet *list*. Il existe deux façons de trier une liste :

- Utiliser la méthode *sort()* : elle s'applique à la liste à trier mais ne **renvoie aucune valeur**.
- La fonction *sorted(liste)* : elle renvoie une autre liste qui est celle passée en paramètre triée.
  Attention : **la liste initiale n'est pas triée.**

Tri de liste avec le langage Python :

```
liste1 = [4,6,1,2]

liste2 = sorted(liste1) # Liste2 contient liste triée
print("liste2 :", liste2) # Liste2 est triée ...
print("liste1 :", liste1) # mais pas liste1 !

liste1.sort() # liste1 triée
print("liste1 triée :", liste1)

liste2.sort(reverse = True) # Liste2 triée dans l'ordre décroissant
print("liste2 décroissant :", liste2)
```

```
liste2 : [1, 2, 4, 6]
liste1 : [4, 6, 1, 2]
liste1 triée : [1, 2, 4, 6]
liste2 décroissant : [6, 4, 2, 1]
```

1/ Principe de la dichotomie

A chaque étape, on partage le tableau trié en deux et on effectue un test pour savoir si l'élément cherché est dans la première partie du tableau (valeur inférieure à la valeur médiane) ou dans la seconde (valeur supérieure à la valeur médiane). On boucle jusqu'à trouver (ou non) la valeur. Il s'agit ici du principe *diviser pour régner* que l'on verra plus loin dans cette partie.

Cette méthode est très efficace car elle divise par deux la taille du tableau par tour.

*Nombre de tours de boucle maximum (**coût** de la recherche)*

Taille de la liste	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**N**
Coût (recherche séquentielle)	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**N**
Coût (recherche dichotomique)	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**$\log_2(N)$**

A noter : Pour tout nombre réel *k*, on a par définition $\log_2(2^k) = k$

Voici un exemple d'algorithme de recherche dichotomique traduit en langage Python :

```
# Cette fonction indique si un element est dans une liste
# (triée) ou non

def rechercheDichotomique(liste,element) :
    indMin = 0
    indMax = len(liste) - 1

    while indMin <= indMax :
        # Recherche de l'indice du centre de l'intervalle
        # l'opérateur "//" assure que l'on obtienne un entier
        indCentre = (indMin + indMax) // 2

        # Récupération de la valeur associée à cet indice
        valCentre = liste[indCentre]

        if valCentre == element :  # Si cette valeur correspond ..
            return True
        elif valCentre < element : # Sinon si elle est inférieure, on décale indMin
            indMin = indCentre + 1 # à indCentre + 1 (évite les effets de bord)
        else :                     # Sinon on décale indMax à indCentre - 1
            indMax = indCentre - 1 # (évite les effets de bord)

    return False

liste1 = [2,3,5,8,10,13,15,20,57]
print(rechercheDichotomique(liste1,10))  # On s'attend à "True"
print(rechercheDichotomique(liste1,17))  # On s'attend à "False"
```

```
True
False
```

2/ Déterminer une solution approchée d'une équation du type *f(x)* = 0

L'algorithme est proche du précédent. Il faut toutefois s'assurer qu'entre les bornes de l'intervalle initial de recherche de la solution que la fonction *f* soit **continue** (pas de « trou » dans le tracé de la courbe représentative de la fonction *f*) et strictement **monotone** (strictement croissante ou décroissante).

Lien vidéo ici : https://www.youtube.com/watch?v=V7mlMCSrq1U
Auteur : Yvan Monka, durée : 21 min 01 sec. Les dix premières minutes expliquent le principe de dichotomie sur un exemple.

Pour généraliser l'exemple d'Yvan Monka (qui a choisi une fonction croissante), on peut proposer l'algorithme suivant, traduit en langage Python :

```python
from math import sqrt

# Définition d'une fonction
# ici, recherche d'une valeur approchée de racine de deux
def fonction(x) :
    return x**2 - 2

# Cette fonction recherche une solution approchée
# d'un équation du type f(x)=0 à epsilon près
def rechercheSolution(borMin,borMax,epsilon) :
    while borMax - borMin > epsilon :
        # Détermination du centre de l'intervalle (nombre réel ici)
        centre = (borMin + borMax) / 2

        # Si le produit des images de bornMin et centre par fonction est négatif,
        # on est sûr que la solution est dans l'intervalle [borMin,centre]
        if fonction(borMin)*fonction(centre) <= 0 :
            borMax = centre
        # sinon elle est dans l'intervalle [centre,borMax]
        else :
            borMin = centre

    return (borMin + borMax)/2

epsilon = 0.001
print("Solution approchée à", epsilon, "près :",rechercheSolution(1,2,epsilon))
print("Racine de 2 : ",sqrt(2))
```

```
Solution approchée à 0.001 près : 1.41455078125
Racine de 2 :  1.4142135623730951
```

Des bibliothèques sont disponibles en Python pour résoudre des équations de manière approchée, comme par exemple celle-ci qui utilise la méthode de dichotomie :

```python
import scipy.optimize

def fonction(x) :
    return x**2 - 2

# La valeur cherchée est entre 1 et 2, fonction est bien
# monotone dans cet intervalle.
borMin = 1
borMax = 2
print("Racine de 2:",scipy.optimize.bisect(fonction,borMin,borMax))
```

```
Racine de 2: 1.4142135623715149
```