TD 16 -
Dichotomie pour :
la recherche de racine dans un tableau
la recherche d'éléments dans une liste triée

Informatique
MPSI-PCSI - Lycée Thiers

Exercice 1

1. Compléter le code suivant d'un fonction `dichotomie` qui prend en paramètres :

– Deux nombres `a` et `b`, avec `a<b`,
– Le tableau `Y` des valeurs prises par une fonction f (inconnue) au dessus de points régulièrement espacés de l'intervalle $[a, b]$.
et qui renvoie :
– Un encadrement le plus fin possible d'une racine de f.
Pour cela on appliquera une recherche par dichotomie. Lorsque la dichotomie est impossible, la fonction renverra un message d'erreur.

Exercice 1

```
def dichotomie(Y,a,b):
    if Y[0]*Y[-1] >0:
        return "Une dichotomie ne s'applique pas"
    debut = 0
    fin = len(Y)-1
    while fin - debut > 1:
        milieu = ....................# Indice du milieu
        if ......................:# Dichotomie
            ..................# à gauche
        else:
            ..................# à droite
    dx = (b-a)/(len(Y)-1)
    return ................., ..................# Encadrement
```

2. La tester sur le tableau Y suivant, entre a=2 et b=4 :

```
import numpy as np
X = np.linspace(2,4,100000)
Y = np.sin(X)
```

Exercice 1. Corrigé

1.

```
def dichotomie(Y,a,b):
    if Y[0]*Y[-1] >0:
        return "Une dichotomie ne s'applique pas"
    debut = 0
    fin = len(Y)-1
    while fin - debut > 1:
        milieu = (debut + fin)//2
        if Y[debut] * Y[milieu] <= 0:
            fin = milieu
        else:
            debut = milieu
    dx = (b-a)/(len(Y)-1)
    return a+dx*debut, a+dx*fin
```

2. On obtient un encadrement du nombre π :

```
In [2]: dichotomie(Y,2,4)
(3.141591415914159, 3.141611416114161)
```

Exercice 2

Exercice 1.

1. Ecrire une fonction `recherche(L,e)` prenant en paramètre un nombre `e` et une liste `L` et qui recherche si `e` apparaît ou non dans la liste `L`. La fonction retournera le booléen `True` ou `False`.

   1.1 en utilisant l'instruction `e in L`.

   1.2 en utilisant une boucle `for x in L:`.

Exercice 2

2. Ecrire une fonction `dich_search()` qui recherche par dichotomie si un élément est présent dans une liste de nombres ordonnée dans le sens croissant.

   2.1 A l'aide d'un slicing. On s'inspirera du code suivant que l'on complétera :

```
def dich_search(L,e):
    while len(L)>1:          # tant que liste contient >1 élément
        iMed = len(L)//2     # indice de l'élément médian
        if ..........:         # Cas de succés
            return True
        elif ...........:
            L=L[iMed:]         # dichotomie à droite
        else:
            L= L[:....]         # dichotomie à gauche
    if (len(L)==1 and ........): # lorsqu'il reste 1 élément
        return .........
    return ..........             # Cas d'échec
```

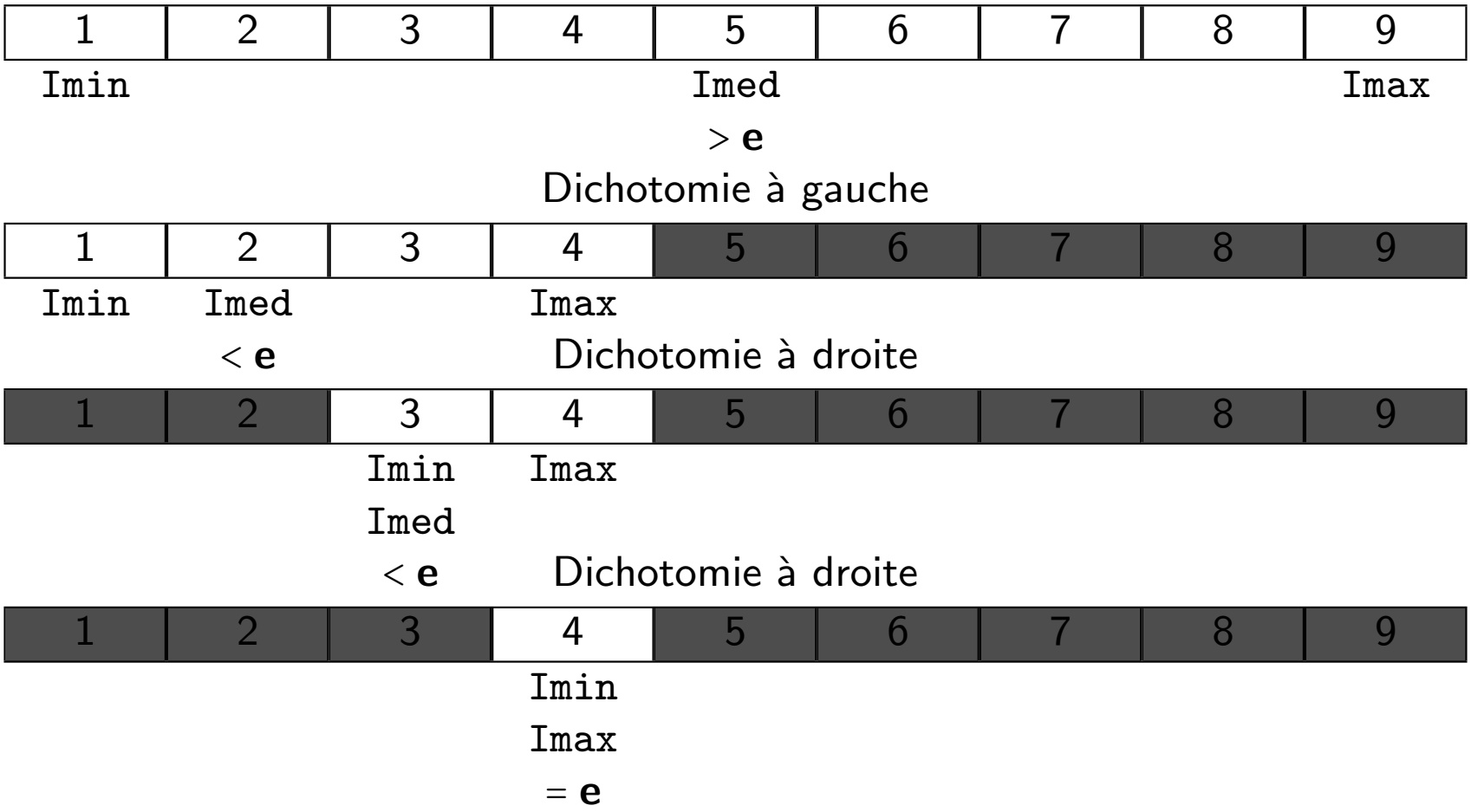
Exercice 2

2. 1.1 Sans utiliser de slicing. On s'inspirera du code suivant que l'on complétera :

```
def dich_search2(l,e):
    Imin, Imax = 0, len(l)-1    # Imin, Imax : délimiteurs début-fin
    while Imax - Imin >= 0:
        Imed = ...........
        if ..........:        # Cas de succès
            return True
        elif ..........:
            Imin = Imed + 1    # Dichotomie à droite
        else:
            Imax = Imed - 1    # Dichotomie à gauche
    return ...........    # Cas d'échec
```

Utiliser deux variables `Imin` et `Imax` de type entiers qui délimitent la partie du tableau à inspecter. `Imed` est l'indice du milieu :

Exemple : Recherche de **e** = **4** dans le tableau suivant :



Exercice 2

3 Tester le temps d'exécution des 4 algorithmes de recherche sur une liste aléatoire ordonnée de 100 000 nombres, grâce aux commandes suivantes (que l'on complétera) :

```
from random import random
L = [random() for k in range(100000)]    # Liste aléatoire
L.sort()    # Tri de la liste

from time import clock
for f in (recherche, recherche2, dich_search, dich_search2):
    a = clock()
    f(L,0)
    b = clock()
    print("avec",f,":",b-a, 'secondes')    # Temps d'éxécution
```

Faire de même avec une liste de 10^6 éléments. Que constate-t-on ?

Exercice 2 : Réponse

```
# Sans l'instruction in
def recherche(l,e):
    for x in l:
        if x == e:
            return True
    return False

# Avec l'instruction in
def recherche2(l,e):
    return e in l
```

Exercice 2 : Réponse

```
def dich_search(l,e):
# Recherche dichotomique dans une liste croissante
    while len(l)>1:          # tant que liste contient >1 élément
        i = len(l)//2    # indice de l'élément médian
        if l[i] == e:          # Cas de succés
            return True
        elif l[i] < e:
            l=l[i:]          # dichotomie à droite
        else:
            l=l[:i]          # dichotomie à gauche
    if (len(l)==1 and l[0]==e): # lorsqu'il reste 1 élément
        return True
    return False              # Cas d'échec
```

Exercice 2 : Réponse sans slicing

2.2

```
def dich_search2(l,e):
    Imin, Imax = 0, len(l)-1
    while Imax - Imin >= 0:
        Imed = (Imin + Imax)//2
        if l[Imed] == e:
            return True
        elif l[Imed] < e:
            Imin = Imed + 1
        else:
            Imax = Imed - 1
    return False
```

Exercice 2.3

```
from random import random
from time import clock
L = [random() for k in range(100000)]
L.sort()

R = []
for f in (recherche, recherche2, dich_search, dich_search2):
    a = clock()
    f(L,0)
    b = clock()
    print("avec",f,":",b-a, 'secondes')    # Temps d'éxécution
    R.append(b-a)
print("Rapport entre plus lent et plus rapide :",max(R)/min(R))
# liste de 10^6
L = [random() for k in range(1000000)]
L.sort()
R = []
for f in (recherche,recherche2,dich_search,dich_search2):
    a = clock()
    f(L,0)
    b = clock()
    print("Avec ",f,":",b-a,"secondes")
    R.append(b-a)
print("Rapport entre plus lent et plus rapide :",max(R)/min(R))
```

Exercice 2.3

```
Avec <function recherche at 0x109e9ed08> : 0.008862999999998067 secondes
Avec <function recherche2 at 0x109e9ec80> : 0.011058999999999486 secondes
Avec <function dich_search at 0x109e9ed90> : 0.0017500000000012506 secondes
Avec <function dich_search2 at 0x10990af28> : 1.9000000001767603e-05 secondes
Rapport entre plus lent et plus rapide : 582.052631524771
Avec <function recherche at 0x109e9ed08> : 0.13202099999999817 secondes
Avec <function recherche2 at 0x109e9ec80> : 0.18923099999999948 secondes
Avec <function dich_search at 0x109e9ed90> : 0.05285699999999949 secondes
Avec <function dich_search2 at 0x10990af28> : 1.6000000002236447e-05 secondes
Rapport entre plus lent et plus rapide : 11826.937498346822
```

On constate qu'une recherche par dichotomie est considérablement plus rapide !

- Sans dichotomie la plus rapide est celle utilisant l'instruction `in` : l'algorithme est le même ; mais il est implémenté à plus bas niveau (langage C), et donc plus rapide : une fonction prédéfinie même si elle fait la même chose qu'un algorithme que l'on écrira le fait mieux et plus vite (en python !).
- Avec dichotomie la recherche est plus rapide sans slicing qu'avec slicing.

Chaque slicing nécessite une copie de la moitié de la liste qui est couteuse en temps (et en mémoire).

Sans slicing on apprécie pleinement les avantages de rapidité de la recherche par dichotomie :

- Sans dichotomie, dans le pire des cas (celui où l'élément ne figure pas dans liste), on doit parcourir toute la liste pour s'en rendre compte : 100000 élément à parcourir (nombre d'itérations de la boucle `for`).
- Avec dichotomie, dans le pire des cas, la boucle `while` est itérée $\log_2(100000) < 17$ fois.
- L'amélioration est encore plus nette avec 10^6 éléments.