1 Dichotomie sur un tableau pour la recherche de racine

Exercice 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.

```
def dichotomie(Y,a,b):
    if Y[0]*Y[len(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
```

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

```
from math import sin
N = 100000
a, b = 2, 4
X = [ a+k*(b-a)/N for k in range(N+1) ]
Y = [ sin(x) for x in X ]
```

2 Recherche d'un élément dans une liste

Il est très souvent nécessaire de déterminer si une valeur apparaît au sein d'une liste. Il faut savoir programmer une fonction qui résout ce problème.

Exercice 2.
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 itérera une recherche au sein d'une boucle **`for`** et retournera le booléen `True` ou `False` selon si `e` apparaît ou non (comme valeur) dans la liste `L`.

Remarque. Le temps d'exécution de cette fonctions de recherche est au plus de l'ordre de la longueur de la liste. Pour une liste 1000 fois plus longue le temps d'exécution sera au pire 1000 fois plus long.

3 Recherche par dichotomie au sein d'une liste triée

3.1 Recherche dans une liste triée

Pour rechercher un élément au sein d'une liste triée (sens croissant par exemple), il est possible d'aller beaucoup plus vite. Dejà, inutile de poursuivre la recherche dès qu'on a atteint un élément strictement plus grand que la valeur recherchée :

```
def recherche_c(L,e):
   for x in L:
      if e == x:
         return True
      elif e < x:
         return False
   return False
```

C'est mieux, mais le temps d'exécution est encore au pire proportionnel à la longueur de la liste. On peut faire beaucoup mieux, grâce à une recherche par dichotomie :

3.2 Recherche par dichotomie dans une liste triée

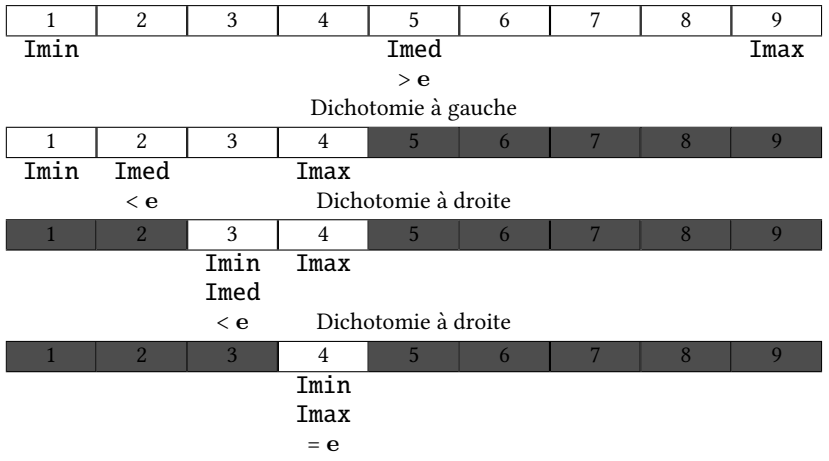
Principe de la recherche par dichotomie :

Donnés une liste `L` triée dans le sens croissant, et une valeur `e` à rechercher dans la liste :

- Comparer l'élément recherché `e` avec l'élément en milieu de liste `m = L[len(L)//2]`.
- En cas d'égalité : l'élément est trouvé : sortie.
- Si `e<m` recommencer sur la première moitié de la liste.
- Si `e>m` recommencer sur la deuxième moitié de la liste.

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 $\mathbf{e} = \mathbf{4}$ dans le tableau suivant :



3.3 Implantation

Exercice 3. Ecrire une fonction `dich_search()` qui recherche par dichotomie si un élément est présent dans une liste de nombres ordonnée dans le sens croissant.

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

```
def dich_search_s(L,e):
    while len(L)>1:     # tant que L contient >1 element
        iMed = len(L)//2    # indice de l'element median
        if ..........:    # Cas de succes
            return True
        elif ...........:
            L=L[iMed:]   # dichotomie à droite
        else:
            L= L[:....]      # dichotomie à gauche
        if (len(L)==1 and ........):     # reste 1 element
            return .........
    return ..........     # Cas d'echec
```

b) Sans slicing. On s'inspirera du code suivant que l'on complétera :

```
def dich_search(L,e):
    Imin, Imax = 0, len(L)-1    # Imin, Imax : delimiteurs
    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'echec
```

c) Tester le temps d'exécution des 4 algorithmes de recherche sur une liste aléatoire ordonnée de 100 000 de nombres, grâce aux commandes suivantes :

```
from random import random
L = [random() for k in range(100000)]    # Liste aleatoire
L.sort()     # Tri de la liste
%
from time import time
for f in (recherche, recherche_c,dich_search_s, dich_search):
   a = time()
   f(L,0.5)
   b = time()
   print("avec",f,":",b-a, 'secondes')  # Temps d'execution
```

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

3.4 Explication

• Pour une liste de taille N le temps d'exécution pour rechercher un élément par dichotomie est du même ordre que la profondeur de la dichotomie (chaque étape ayant une durée d'exécution bornée).

• Alors pour une liste de N éléments combien est au plus la profondeur de la dichotomie, pour trouver un élément, en fonction de N ?

• Il est plus facile de calculer la fonction réciproque : Si pour une liste au plus k profondeurs de dichotomies sont nécessaires pour chercher un élément, combien d'éléments contient cette liste approximativement ?

– À chaque dichotomie on sépare la liste en 2 listes de tailles égales ± 1.
– Donc si à l'étape $i+1$ la liste contient N_{i+1} éléments, alors à l'étape i la liste contient entre $2 \times N_{i+1}$ et $2 \times (N_{i+1}+1)$ éléments .
– À la dernière étape, dans le pire des cas, la liste ne contient plus qu'un élément.

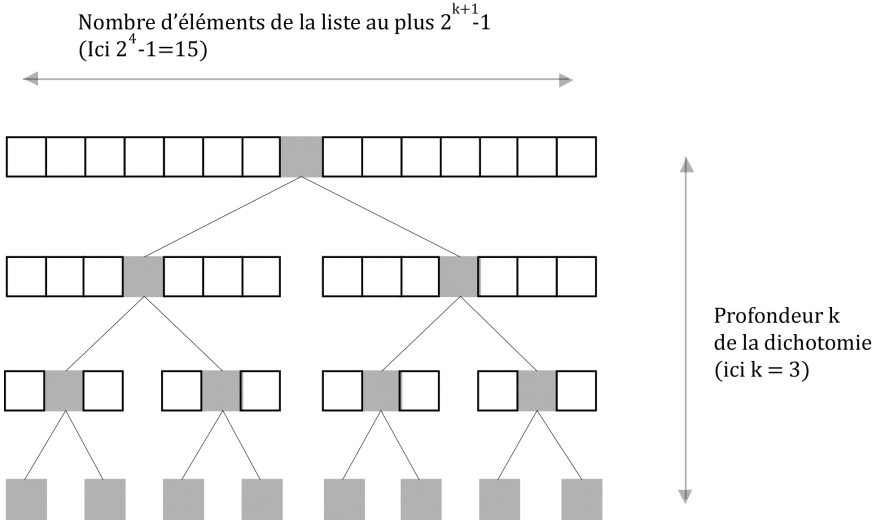
Ainsi la liste était initialement de taille comprise entre 2^k et au plus :

$$\sum_{i=0}^{k} 2^i = \frac{1-2^{k+1}}{1-2} = 2^{k+1} - 1$$

• Donc en composant par $\log_2$ (qui est croissante) :

$$\begin{aligned} 2^k \leqslant N \leqslant 2^{k+1} &\Longrightarrow k \leqslant \log_2(N) \leqslant k+1 \\ &\Longrightarrow \log_2(N) - 1 \leq k \leq \log_2(N) \end{aligned}$$

La profondeur de dichotomie k a pour ordre $\log_2(N)$.



• Les cases grisées représentent les seuls éléments susceptibles d'être comparés avec l'élément recherché aux diverses étapes de la dichotomie.

Lors d'une implémentation, seules celles sur une branche (jusqu'en bas dans le pire des cas) seront comparées, alors que l'algorithme naïf peut comparer avec tous les éléments de la liste.

• Le temps d'exécution d'une recherche par dichotomie dans une liste triée dans le sens croissant est au pire de l'ordre de

$\boxed{\log_2(N)}$ **(on parle de complexité logarithmique)**

• C'est beaucoup plus rapide par dichotomie que par une recherche naïve, dont le temps d'exécution est de l'ordre de N (on parle de complexité linéaire) :