Recherche dichotomique dans un tableau trié 💔

Introduction : un petit jeu

En début d'année, nous avons programmé ce petit jeu :

```
from random import randint                                    ###
nb_mystere = randint(0,100)

print("Esssayez de trouver le nombre mystère")
nb_essais =0
trouvé = False

while not(trouvé):
    choix = int(input("Entrer un nombre entre 0 et 100 :"))
    nb_essais += 1
    if choix > nb_mystere :
        print("C'est trop grand")
    elif choix < nb_mystere :
        print("C'est trop petit")
    else :
        trouvé = True

print("Bravo ! Nombre de tentatives = ", nb_essais)

```

```
>>>
```

Vous pouvez y rejouer en copiant le code dans un IDE ou dans une console Basthon.

Quelle stratégie vous semble la plus pertinente pour trouver le nombre le plus rapidement possible ?

1- Rappel : Recherche séquentielle

Le tableau est une structure de données extrêmement utilisée en informatique.

Une des questions que l'on peut se poser est de savoir si une valeur appartient, ou non, à ce tableau.

Nous avons déjà vu l'algorithme de **recherche séquentielle**.

Algorithme de recherche séquentielle

```
def recherche_sequentielle(tab, val):                                    ###
    """
    Entrées :
        tab est un tableau
        val est une variable de même type que les éléments du t
    Sortie : un booléen qui vaut True si val appartient au tabl
    """
    for k in range(len(tab)):
        if tab[k] == val:
            return True
    return False

# Test de la recherche séquentielle
from random import randint

taille = 50
tableau = [randint(1, taille) for k in range(taille)] # un tabl
print("Tableau :", tableau)
valeur = randint(1, taille) # un nombre choisi aléatoirement
print("Valeur :", valeur)

print(recherche_sequentielle(tableau, valeur))

```

```
>>>
```

L'algorithme précédent est **correct** : on peut prouver qu'il fonctionne correctement dans tous les cas.

Nous avons admis que, comme l'algorithme de calcul de moyenne ou celui de calcul d'un minimum/maximum, il est de **complexité linéaire** : le nombre moyen d'instructions effectuées et le temps moyen de

calcul sont proportionnels à la taille du tableau.

Vérification expérimentale de la performance de la recherche séquentielle

On peut vérifier expérimentalement que si la taille d'un tableau est multipliée par 100, alors le temps de recherche séquentielle d'une valeur dans ce tableau est aussi multiplié par 100 (*approximativement*).

De préférence, copier et exécuter ce code dans un IDE.

```
1  from random import randint                                    ###
2  from time import perf_counter
3
4  for taille in [10**4, 10**6]:
5      tab = [randint(1, taille) for k in range(taille)]
6      val_alea = randint(1, taille)
7
8      debut = perf_counter()
9      recherche_sequentielle(tab, val_alea)
10     fin = perf_counter()
11
12     print("Pour un tableau de taille n =", taille, "\n il faut
13
```

```
>>>
```

2 - Principe de la recherche dichotomique

Le fait qu'un tableau soit trié, par exemple par ordre croissant, facilite de nombreuses opérations.

Le fait de disposer d'un tableau déjà trié se produit par exemple quand on dispose d'enregistrements chronologiques, comme ci-dessous dans le tableau des décollages d'avions dans un aéroport.

🐍 **Script Python**

```
tab_departs = [(12.39, 'Londres'), (12.57,'Zurich'), (13.08,'Dublin'),
(13.21,'Casablanca'),
               (13.37,'Amsterdam'),(13.48,'Madrid'),(14.19,'Berlin'), (14.35,'New
York'),
              (14.54, 'Rome'), (15.10,'Stockholm')]
```

Supposons que l'on cherche à savoir si un décollage est prévu à 14h00 pile.

On peut lancer une recherche séquentielle du nombre `14.00` dans ce tableau.

Mais on peut **tirer profit du fait que le tableau est déjà ordonné, déjà trié**. Voici un procédé possible :

1. On commence par comparer la valeur recherchée (ici `14.00`) avec la valeur située au milieu du tableau (ici `13.37`) :
2. Si la valeur recherchée est plus petite, on peut restreindre la recherche à la première moité du tableau.
3. Sinon, on la restreint à la seconde moitié du tableau.
4. Et on recommence, mais avec une partie du tableau deux fois plus petite. À chaque étape, on divise la zone de recherche par deux à chaque étape : très rapidement, on parviendra soit à la valeur recherchée, soit à un intervalle vide.

On appelle ceci une **recherche dichotomique** (en anglais : *binary search*).

> **Définition du mot *dichotomie* du dictionnaire Larousse**
>
> Division de quelque chose en deux éléments que l'on oppose nettement. Exemple : Dichotomie entre la raison et la passion.

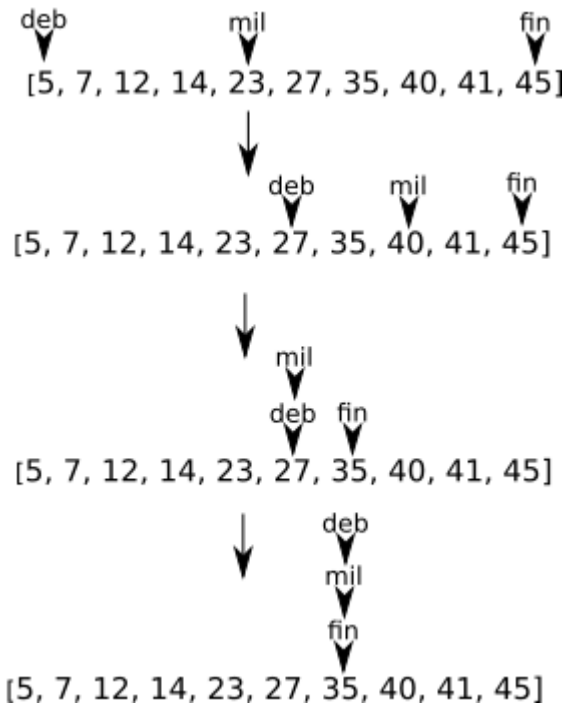
Dans notre procédé de recherche par dichotomie, on fait une **division du tableau trié en deux parties** : d'un côté la première moitié (composée des petites valeurs), de l'autre la seconde moitié (composée des grandes valeurs).

3 - La méthode dichotomique, par l'exemple

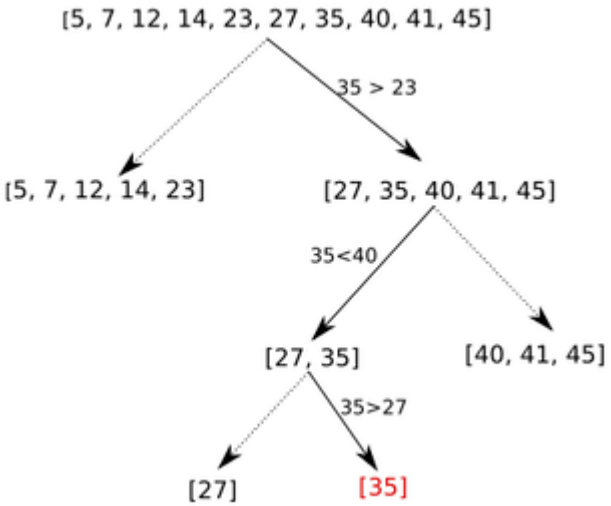
Supposons que l'on recherche si la valeur **35** est présente ou non dans le tableau trié suivant : `[5, 7, 12, 14, 23, 27, 35, 40, 41, 45]`

Les étapes de recherches sont schématisées ci-dessous, avec :

- **debut** qui marque le **début de la zone de recherche**.
- **fin** qui marque la **fin de la zone de recherche**.
- **milieu**, la **valeur médiane entre le début et la fin de la zone** de recherche. Cette valeur médiane `milieu` est le résultat du calcul de la moyenne entre `debut` et `fin`, *arrondie si besoin à l'entier inférieur* : `milieu = (debut + fin) // 2`



Si vous préférez, voici une autre **représentation de cette recherche, sous la forme d'un arbre** :



Exercice

Représenter les différentes étapes de la recherche dichotomique de la valeur 7 dans le tableau `[0, 1, 1, 2, 3, 6, 8, 12, 21]`

4 - Algorithme de recherche dichotomique d'un tableau trié

Remarques préliminaires :

- en entrée de l'algorithme, on fournit un tableau `tab` qui est supposé **trié par ordre croissant**, ainsi qu'une valeur `val` à rechercher ;
- en sortie, l'algorithme doit renvoyer `True` si la valeur `val` est dans le tableau `tab` et `False` sinon.

Voici en pseudo-code l'algorithme de recherche par dichotomie :

📋 **Pseudo-code**

```
FONCTION Recherche_Dichotomique(tab,val)
    debut ← 0  # choix de numéroter à partir de zéro les cases du tableau
    fin ← longueur(tab) - 1
    TANT_QUE debut ⩽ fin
        milieu ← (debut + fin) // 2
        SI tab[milieu] = val ALORS
             RENVOYER VRAI
        SINON
            SI val > tab[milieu] ALORS
                debut ← milieu + 1
            SINON
                fin ← milieu - 1
            FIN_SI
        FIN_SI
    FIN_TANT_QUE
    RENVOYER FAUX
```

Exercice

Écrire le code Python de cet algorithme de recherche dichotomique.

```
def recherche_dichotomique(tab, val):                                    ###
    """
    Entrées :
        tab est un tableau de nombres trié par ordre croissant
        val est un nombre
    Sortie : un booléen qui vaut True si val appartient au tableau
    """
    # compléter le code

```

```
>>>
```

Testons alors cet algorithme :

```
from random import randint                                               ###

taille = 50
tableau = sorted([randint(1, taille) for k in range(taille)])
print("Tableau :", tableau)
valeur = randint(1, taille)
print("Valeur :", valeur)

print(recherche_dichotomique(tableau, valeur))

```

```
>>>
```

5 - Efficacité, complexité de la recherche dichotomique

L'algorithme de recherche élaboré ci-dessus applique le principe ***"diviser pour régner"*** : à chaque étape (si la valeur n'a pas encore été trouvée), la charge de travail est **divisée par deux**.

Inversement, si on multiplie par 2 la taille du tableau, il n'y aura besoin que d'une étape supplémentaire pour y effectuer la recherche d'une valeur.

Puisqu'il ne faut qu'une étape pour faire une recherche dans un tableau de taille 2, on peut dresser le tableau suivant :

Taille du tableau	**2**	**4**	**8**	**16**	**32**	**64**
Nombre maximal d'étapes	1	2	3	4	5	6

On voit ainsi apparaitre le lien mathématique entre ces deux quantités : la taille t du tableau est égale à $ 2^n$ où n est le nombre de d'étapes de l'algorithme.

Or ce que l'on cherche à connaitre, c'est la relation "inverse", à savoir le nombre d'étapes n en fonction de la taille t. En mathématiques, on parle de "fonction réciproque" et la fonction réciproque de la fonction ***"2 à la puissance n***" s'appelle la fonction ***logarithme de base 2***.

Elle se note $\log_2$ (en mathématiques) et `log2` en Python (avec le module `math`) : on a donc $\log_2(8) = 3$, $\log_2(16) = 4$, etc.

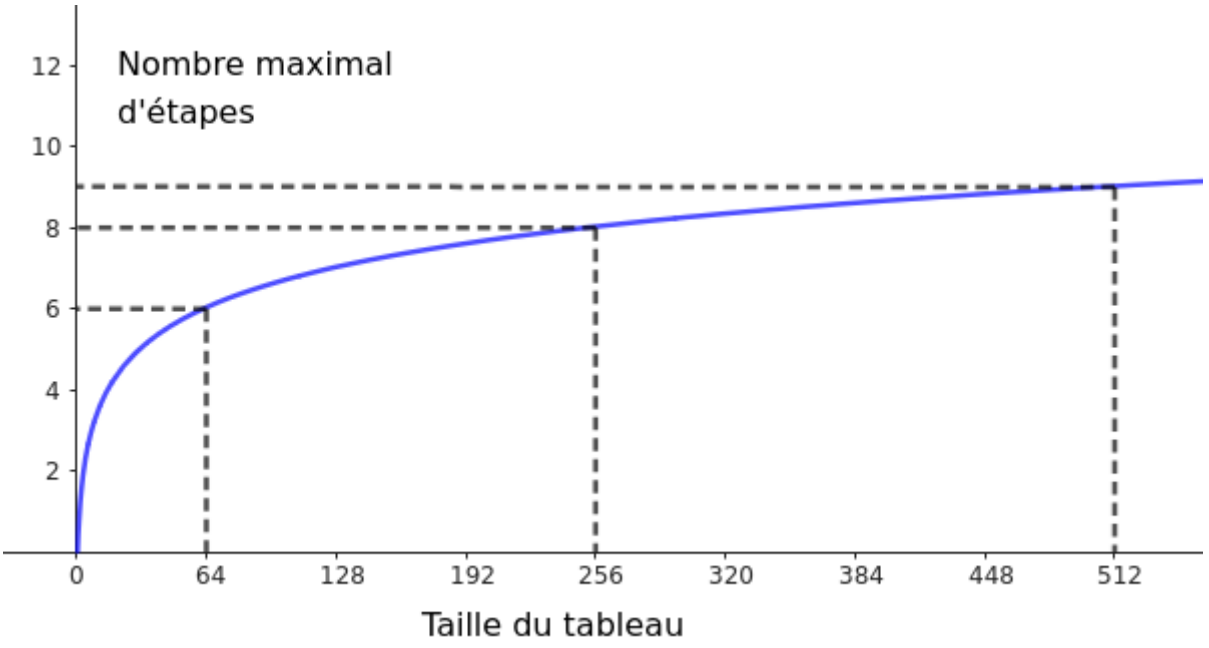
```
1  from math import *                                    ###
2  print("logarithme de base 2 de 16 : ", log2(16))
3
```

```
>>>
```

Complexité logarithmique

Le nombre maximal d'étapes de l'algorithme de recherche par dichotomie s'écrit alors $n = \log_2(t)$, où t est la taille du tableau. On dit que la recherche dichotomique est un algorithme de **complexité logarithmique**.

Il est donc **extrêmement efficace**, bien plus efficace qu'un algorithme de complexité linéaire.



On rencontrera parfois l'expression "complexité en $O(\log_2(t))$".

Mais n'oublions pas que l'utilisation d'une recherche dichotomique nécessite que le tableau soit trié, et que le tri d'un tableau est une opération coûteuse (en temps). ***On ne doit donc utiliser cet algorithme que dans le cas où l'on dispose d'un tableau déjà trié.***

6 - À retenir

1. Une recherche dichotomique ne peut se faire que sur un tableau trié.

2. Une recherche dichotomique consiste à systématiquement découper la zone de recherche en deux jusqu'à trouver (ou non) la valeur cherchée :
    - La zone de recherche est délimitée par un indice de début et un indice de fin.
    - On teste si la valeur médiane de cette valeur de recherche est égale à la valeur cherchée.
    - Tant que l'on n'a pas trouvé la valeur cherchée, on restreint la zone de recherche en déplaçant l'indice de début ou l'indice de fin.
    - Si, à l'issue de ces redécoupages successifs, la zone de recherche se réduit à une seule valeur et qu'on a toujours pas trouvé la valeur cherchée, c'est que la valeur est absente du tableau.

7 - Exercices

Exercice 1

Adapter le programme de recherche dichotomique pour écrire une fonction `recherche_comptage(tab,val)` qui :

- prend en entrées un tableau `tab` trié par ordre croissant et d'une valeur `val`
- renvoie un couple (*tuple*) `(trouvé,nb_tours)` composé
  - d'un booléen `trouvé` qui vaut `True` si la valeur `val` appartient au tableau et `False` sinon
  - du nombre entier `nb_tours` égal au nombre de tours de la boucle `while` qu'il a fallu effectuer dans cette recherche dichotomique.

Ne pas oublier de tester votre fonction.

```
1 ###
2
3
```

```
>>>
```

Exercice 2 : recherche dichotomique dans un tableau de couples

Adapter l'algorithme de recherche dichotomique pour pouvoir traiter le tableau de couples de la partie 2, où le premier élément des couples est l'heure de décollage d'un avion et où le tableau est déjà trié par ordre croissant d'horaire.

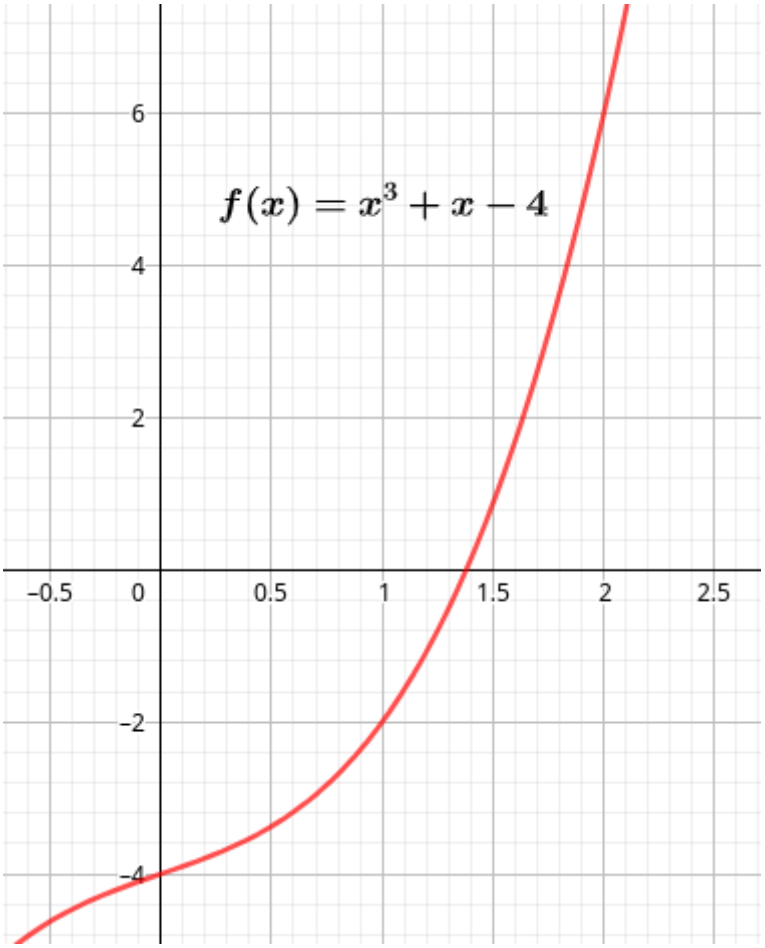
```
1  def recherche_dichotomique_couple(tab,val):                    ###
2      """
3      Entrées :
4          tab est un tableau de couples (nombre, texte), trié par
5          val est un nombre
6      Sortie : un booléen qui vaut True si val est un nombre d'un
7      """
8      # compléter le code
9
10 # test de la fonction
11 tab_departs = [(12.39, 'Londres'), (12.57,'Zurich'), (13.08,'Du
12                 (13.37,'Amsterdam'),(13.48,'Madrid'),(14.19,'Ber
13                (14.54, 'Rome'), (15.10,'Stockholm')]
14 assert recherche_dichotomique_couple(tab_departs,14.00) == Fals
15
```

```
>>>
```

Exercice 3 : une autre application de la dichotomie

Quand on trace la fonction f définie par la formule $f(x) = x^3 + x - 4$, on constate que :

- $f(1) < 0$ et $f(2) > 0$
- la fonction s'annule une fois entre 1 et 2.



On veut trouver **une valeur approchée, au millionième près, de ce nombre inconnu** x_0 qui vérifie $f(x_0) = 0$.

1°) On commence par définir en Python la fonction `f` :

Script Python

```
def f(x):
    y = x**3 + x-4
    return y
```

2°) Puis on va procéder par dichotomie :

On définit les variables :

- **debut** , qui marque le début de la zone de recherche, est égal à 1 au départ ;
- **fin** , qui marque la fin de la zone de recherche, est égal à 2 au départ ;
- **milieu** est la moyenne entre `debut` et `fin` : `(debut + fin) / 2`

Et on applique l'algorithme suivant :

📋 **Pseudo-code**

```
TANT_QUE fin - debut > 1e-6 # précision demandée de un millionième
    milieu ← (debut + fin) / 2
    SI f(milieu) = 0 ALORS
        AFFICHER milieu
    SINON
        SI f(milieu) < 0 ALORS
            debut ← milieu
        SINON
            fin ← milieu
FIN_TANT_QUE
AFFICHER milieu
```

Programmer cet algorithme en Python et déterminer la valeur approchée à 10^{-6} de ce nombre x_0.

Combien de tours de boucle `TANT QUE` ont été nécessaires pour obtenir cette valeur de x_0 ?

```
1
2
3
```

###

>>>

8 - QCM

Partie 1

On considère le code suivant de recherche d'une valeur dans une liste :

Script Python

```
def search(x, tab):
    # x est la valeur à chercher
    # tab est une liste de valeurs
    for k in range(len(tab)):
        if x == tab[k]:
            return True
    return False
```

Question

- Quel est le coût de cet algorithme ?

  - ◯ constant
  - ◯ logarithmique
  - ◯ linéaire
  - ◯ quadratique

Explication

On reconnait ici l'algorithme de recherche séquentielle, qui a un coût proportionnel à la taille du tableau : on parle alors d'un coût ***linéaire***.

Partie 2

Questions

1. Pour pouvoir utiliser un algorithme de recherche par dichotomie dans une liste, quelle précondition doit être vraie ?

   - ◯ la liste doit être triée
   - ◯ la liste ne doit pas comporter de doublons
   - ◯ la liste doit comporter uniquement des entiers positifs
   - ◯ la liste doit être de longueur inférieure à 2^{10}

2. La fonction ci-dessous permet d'effectuer une recherche par dichotomie de l'index `m` de l'élément `x` dans un tableau `L` de valeurs distinctes et **triées**.

   🐍 **Script Python**

   ```
   def dicho(x,L):
       g = 0
       d = len(L)-1
       while g <= d:
           m = (g+d)//2
           if L[m] == x:
               return m
           elif L[m] < x:
               g = m+1
           else:
               d = m-1
       return None
   ```

   Combien de fois la cinquième ligne du code de la fonction (`m = (g+d)//2`) sera-t-elle exécutée dans l'appel `dicho(32, [4, 5, 7, 25, 32, 50, 51, 60])` ?

   - ◯ 1 fois
   - ◯ 2 fois
   - ◯ 3 fois
   - ◯ 4 fois

3. On décide d'effectuer une recherche dans un tableau trié contenant 42000 valeurs.

   On procède par dichotomie.

   Le nombre maximal d'itérations de l'algorithme sera :

   - 21000 car une recherche dichotomique divise le nombre de tests maximal par deux
   - 42000 car la valeur recherchée pourrait très bien être la dernière du tableau
   - 41999 car si on n'a pas trouvé l'élément recherché à l'avant-dernière position du tableau, il n'est plus utile d'effectuer de test pour la dernière position
   - 16 car à chaque itération, le nombre d'éléments à examiner est divisé par deux et que $2^{15} \leqslant 42000 < 2^{16}$

4. Un algorithme de recherche dichotomique dans une liste triée de taille n nécessite, dans le pire des cas, de réaliser k comparaisons.

   Combien cet algorithme va-t-il effectuer, dans le pire des cas, de comparaisons sur une liste de taille $2imesn$?

   - k
   - $k+1$
   - 2 × k
   - 2 ×$(k+1)$