Problème de la recherche d'un élément dans une liste

On étudie un problème récurrent en informatique : trouver un élément dans une liste **ordonnée/triée** (*nous verrons dans un prochain cours comment trier ou ordonner une liste*) : chaque élément est plus grand que le précédent mais plus petit que le suivant.

Approche simpliste du problème

Une façon naturelle de faire serait de regarder le premier élément de la liste, le comparer à celui qu'on recherche :

- ▷ si c'est celui qu'on recherche : jackpot !
- ▷ sinon on passe au suivant
- ▷ si on arrive à la fin de la liste sans l'avoir trouvé, c'est qu'il n'était pas dedans

Application 1 : 🚫🚫🚫 **Attention !** Interdit d'utiliser les tests d'appartenance comme "in" !

En utilisant l'algorithme précédent, définir une fonction *Recherche* qui prend en entrée un élément e et une liste triée L et qui renvoie :

- ▷ *False* si e n'est pas dans L
- ▷ *True* si il s'y trouve

Votre programme fonctionnerait-il avec une liste non-triée ?

On se rend compte que l'algorithme proposé de recherche n'utilise pas le fait que la liste est triée : il est fort probable qu'il ne soit pas optimal !

Application 2 : On travail sur une liste de n éléments.

1. Estimer le nombre de test d'égalité que va devoir réaliser votre programme **dans le pire des cas**.
2. Si chaque test d'égalité prend 0.01ms combien de temps prendra la recherche si :
 - ▷ L contient 1000 élément
 - ▷ L contient 10^{12} éléments

Notion de dichotomie

Exemple 1 : Avec votre voisin, jouer à un jeu : le premier pense à un nombre entre 0 et 100, le second doit deviner lequel. Pour ce faire, il peut proposer un nombre, le premier joueur devra lui dire si le nombre auquel il pense est "plus grand" ou plus petit".
Quelle est l'algorithme optimal ?

L'idée de la recherche par dichotomie est de réduire à chaque fois par deux la taille de l'intervalle (ou de la liste) dans laquelle on va rechercher l'élément.

Principe de la méthode dichotomique :

On propose la fonction de recherche dichotomique suivante :

```
def rechercheDichotomique (L,e):
    '''
    L : liste triée en ordre croissant
    e : élément quelconque
    Return : renvoie True si e est dans L, False sinon
    '''

    a,b=0, len(L)-1
    while a<=b :
        m=(a+b)//2
        if L[m]==e:
            return True
        elif L[m]<e:
            a=m+1
        else :
            b=m-1

    return False
```

► Documentation d'une fonction

A quoi servent les lignes compris entre ''' ... ''' ? Pour le savoir, taper `help(rechercheDichotomique)`. S'affiche alors les information qu'on a mis entre triple guillemets.

🚨🚨🚨 **Attention !** Un bon réflexe à avoir lors d'un projet long est de documenter ses fonction en mettant :

- ▷ les paramètres en entrée de la fonction **dans l'ordre**
- ▷ ce que renvoie la fonction

► Bien comprendre l'architecture d'un algorithme

Appliquer à la **main** ce que fait la fonction `rechercheDichotomique` sur :

- ▷ la liste $L = [1, 3, 34, 56, 78, 99, 104, 423, 566, 1024]$
- ▷ l'élément $e = 104$ puis $e = 32$.

► Complexité de l'algorithme

On ne cherchera pas à démontrer la complexité d'un algorithme dichotomique. Néanmoins on peut la comparer "à la main" à celle d'un algorithme classique :

- ▷ algorithme classique : à chaque étape de la recherche on exclue 1 élément. On réalise N étape jusqu'à tester tous les n éléments de L :

$$N \times 1 = n$$

On a bien un complexité linéaire.

- ▷ algorithme dichotomique : à chaque étape, on divise par deux l'intervalle dans lequel on cherche l'élément. Au début on a $len(L)$, puis $len(L)/2$, ... On appelle $n = len(L)$:

$$\frac{n}{2} \rightarrow \frac{n}{4} \rightarrow \frac{n}{8} \rightarrow \dots \text{ soit } \frac{n}{2^N} = 1 \Rightarrow N = \frac{\ln n}{\ln 2}$$

On a une complexité logarithmique.

Propriété. Complexité logarithmique

Un algorithme dichotomique possède une complexité logarithmique. Le nombre d'opération élémentaires nécessaires n pour terminer l'algorithme évolue comme :

$$\simeq \frac{\ln n}{\ln 2}$$

avec n la taille des données.

Application 3 : Linéaire versus logarithmique

On souhaite comparer deux algorithmes, un de complexité linéaire et l'autre logarithmique. Si on multiplie par 10 la taille des données en entrée, par combien est multiplié le temps de calcul pour chacun des deux algorithmes ?

► Application : recherche d'un élément avec une précision

A partir de la fonction *rechercheDichotomique* donnée, écrire la fonction *rechercheDichotomiqueApprochee* qui :

- ▷ prend en entrée une liste ordonnée de nombres L et deux nombres x et δx
 - ▷ renvoie *True* s'il existe un élément y de L tel que $y \in [x - \delta x, x + \delta x]$. Elle renvoie *False* sinon.
- Modifier la fonction pour qu'elle renvoie également la position de l'élément y dans la liste L et 0 si l'élément ne s'y trouve pas..

Recherche du zéro d'une fonction

Un exemple classique de recherche d'un élément dans une liste est de trouver les points d'annulation d'une fonction. On introduit pour cela une **précision** δx .

Soit une fonction f . On cherche x_0 tel que $|f(x_0)| < \delta x$.

♡ *Instant math* ♡

1. Soit un intervalle $[a, b]$. A quelle condition sur f l'intervalle image de $[a, b]$ par f est ordonné ?
On considère par la suite cette condition comme vérifiée.
2. A quelle condition sur $f(a)$ et $f(b)$ la fonction f s'annule sur l'intervalle $[a, b]$.

Fini les maths ...

3. Écrire une fonction *recherche_zero* qui prend en entrée une fonction f , deux nombres $a < b$ tels que f soit croissante sur $[a, b]$, un nombre δx et qui renvoie :
 - ▷ "*False*" si la fonction ne s'annule pas sur l'intervalle $[a, b]$
 - ▷ "*True*" si la fonction s'annule et une estimation du zéro de f à δx près