ALGORITHMIQUE AVEC LES LISTES ET CHAÎNES DE CARACTÈRES

Le but de ce chapitre est d'apprendre à programmer quelques algorithmes élémentaires qui font intervenir les listes et les chaînes de caractères.
Beaucoup de ces algorithmes sont déjà implémentés dans Python. Leur programmation est donc un *exercice* qui n'a d'utilité que pédagogique (et qui vous aidera à programmer d'autres fonctions plus compliquées). Si vous aviez besoin d'utiliser une telle fonction dans un projet, il faudrait utiliser celle de Python si elle existe et ne pas la reprogrammer vous-même.

Ces algorithmes sont des attendus du programme : À l'exception du tri à bulles, vous devez être en mesure de les programmer rapidement un jour d'oral.

1 RECHERCHE DANS UNE LISTE

A Recherche d'un élément quelconque dans une liste

Le but de cette question est de programmer manuellement certaines fonctions de recherche dans les listes.

Fonction "in"
`a in liste` teste l'appartenance de `a`, à la liste `liste`.

```
def appartient(a,liste):
    for i in liste:
        if i == a:
            return True
    return False
```

Python 1: Appartenance à une liste

On parcourt la liste avec une boucle **for**. Dès que l'on rencontre `a`, on arrête pour renvoyer `True`.
Si à la fin de la boucle **for**, on n'a toujours pas trouvé `a`, alors on renvoie `False`.

Remarque : Même programmée ainsi, nous utilisons l'opération **in**, on pourrait s'en passer en utilisant une boucle **while**.

Méthode "index"
Ce serait cet algorithme de recherche que vous devriez savoir refaire s'il n'y en avait qu'un.
On modifie un peu la fonction précédente pour que la nouvelle fonction :

- lève une erreur si `a` n'est pas dans la liste,
- sinon, renvoie le premier indice où se trouve `a`.

```
def indice(a,liste):
    for i in range(len(liste)):
        if liste[i] == a:
            return i
    raise ValueError(str(a)+" n'est pas dans la liste")
```

Python 2: Indice d'un élément dans une liste

Remarques :

- Cette fois-ci la boucle **for** sur l'objet itérable `liste` n'est pas suffisante car on perd la valeur de l'indice. Il faut donc itérer sur les indices avec `range(len(liste))`.
- On aurait pu écrire simplement `raise Exception(...)`, mais on a utilisé `raise ValueError(...)` pour avoir le même type de comportement que la méthode Python.

Méthode "count"
On compte le nombre de fois que l'on trouve `a` dans la liste. Contrairement aux fonctions précédentes, on ne doit pas s'arrêter dès que l'on trouve la valeur, mais parcourir la liste jusqu'au bout pour voir si elle n'y est pas d'autres fois.

```
def compter(a,liste):
    count = 0
    for i in liste:
        if i == a:
            count += 1
    return count
```

Python 3: Nombre d'occurences d'un élément dans une liste

Exercice

Programmez la fonction `indices` qui renvoie la liste de tous les indices où se trouve `a`. Si `a` n'est pas dans `liste`, alors la fonction renvoie la liste vide.

B Recherche du maximum d'une liste

On veut implémenter la fonction `max` pour les listes de nombres. On supposera la liste non vide.

```
def maximum(liste):
    maxi = liste[0]
    for m in liste:
        if m > maxi:
          maxi = i
    return maxi

#Version alternative pour ne pas lire deux fois le premier élément
def maximum(liste):
    maxi = liste[0]
    for m in liste[1:]: #modifié
        if m > maxi:
            maxi = i
    return maxi
```

Python 4: Nombre d'occurences d'un élément dans une liste

Au fur et à mesure que l'on avance dans la liste, `maxi` correspond au maximum de la sous-liste déjà parcourue.

Lorsque l'on est arrivé au bout, `maxi` correspond donc au maximum de la liste complète.

Ce type de raisonnement qui consiste à *"grignoter"* la liste est souvent utilisé. Nous le retrouverons dans le tri des listes.

Cet argument nous permet de prouver formellement par récurrence que notre algorithme se termine et renvoie le bon résultat.

Preuve :

On fait une récurrence sur la longueur de la liste pour montrer que la boucle se termine au bout de n étapes et qu'à l'issue de la boucle, `maxi` vaut bien le maximum de la liste.

- **Initialisation :** pour une liste de longueur 1.
  elle n'a qu'un seul élément `liste=[a]`.
  Au début `maxi` prend la valeur `liste[0]`.
  On rentre dans la boucle : m prend la valeur `a`.
  alors "`m>maxi`" est faux, et on passe à l'incrément suivant.
  Le parcours de la liste se termine ici car elle ne contenait qu'un seul élément, donc on sort de la boucle `for`.
  La boucle s'est donc terminée au bout d'une étape, et `maxi` vaut `a` et qui est bien le maximum.
- **Hérédité :** Soit $n \in \mathbb{N}^*$ quelconque fixé.
  On suppose que pour toute liste de longueur n l'algorithme se termine en n étapes et qu'à l'issue de celles-ci, `maxi` vaut bien le maximum de la liste.

Supposons que `liste` soit une liste de longueur $n+1$, alors on peut l'écrire sous la forme `liste=liste0.append(a)` avec `a` le dernier élément et `liste0` une liste de longueur n.

La boucle parcourt d'abord `liste0`, et par hypothèse de récurrence,

- la boucle a parcouru `liste0` en n étapes,
- lorsque m est à la dernière valeur de `liste0`, `maxi` vaut le maximum de `liste0` par hypothèse de récurrence.

À l'incrément suivant, m vaut a.
Si `a>maxi`, alors `maxi` prend la valeur `a`, sinon `maxi` conserve la même valeur : celle du maximum de `liste0`.
Or le maximum de `liste` est égal au maximum entre `a` et le maximum de `liste0`. C'est bien la valeur de `maxi`.
Et la boucle se termine au bout de $n+1$ étapes.

- **Conclusion :** Par principe de récurrence, pour n'importe quelle longueur de liste, la fonction renvoie bien le maximum de la liste après un calcul ayant autant d'étapes que la longueur de la liste.

■

Remarque : Vous voyez que prouver un algorithme est assez laborieux, c'est la raison pour laquelle on ne le fait pas à chaque fois.

Complexité :

Ce raisonnement nous permet aussi de donner la **complexité** de l'algorithme : c'est-à-dire le nombre d'opérations à réaliser en fonction de la taille de la liste. Pour une complexité, on ne s'intéresse qu'à l'ordre de grandeur.

Ici, il faut n tests, on dira donc que l'on a une complexité en n : *la complexité est **linéaire** par rapport à la taille de la liste.*

Lorsque l'on conçoit un algorithme, le but est d'avoir la complexité la plus faible possible pour que l'algorithme soit rapide.

C Application aux chaînes de caractères

On veut tester si une chaîne de caractère `mot` est incluse dans une autre : `texte`.
Cela revient à programmer la commande **in** pour les chaînes de caractères. La différence avec les listes est qu'il ne suffit pas de tester caractère par caractère, mais avec une chaîne de plusieurs caractères.

```
def appartientStr(mot,texte):
    for i in range(len(texte)-len(mot)+1):
        if texte[i:i+len(mot)] == chaine:
            return True
    return False
```

Python 5: Recherche d'une sous-chaîne

On pourrait améliorer sensiblement cette programmation qui n'est pas du tout efficace dès que `mot` devient un peu long.

Exercice

Programmez une fonction indice et une fonction de comptage comme cela a été fait pour les listes.

2 TRI D'UNE LISTE

Nous allons programmer des algorithmes de tri *en place*. Le terme "*en place*", veut dire que l'on ne va pas créer de nouvelles listes, mais modifier la liste existante peu à peu jusqu'à ce qu'elle soit triée.

L'avantage d'une telle méthode est qu'elle est économe en espace mémoire (on n'a pas besoin de recopier la liste).

A Tri par sélection

C'est le tri le plus simple à programmer. Vous prenez le plus petit élément de la liste et vous le placez au début, puis le deuxième plus petit que vous mettez en deuxième position... jusqu'à ce que tout soit trié.
À l'étape p,

- la sous-liste des p premiers éléments est triée,
- tous les éléments de cette sous-liste sont inférieurs aux restants.

```
def triSelection(liste):
    # modifie en place
    # va chercher le plus petit et le met au début
    for p in range(len(liste)):
        indiceMin = p
        for i in range(p+1,len(liste)):
            if liste[i] < liste[indiceMin]:
                indiceMin = i
        (liste[p],liste[indiceMin]) = (liste[indiceMin],liste[p])
```

Python 6: Tri par sélection

Preuve :

par récurrence en montrant la double propriété : À l'étape p, la sous-liste des p premiers éléments est triée et tous ses éléments sont inférieurs à ceux restants.
Donc à l'étape n, toute la liste est triée.

■

Complexité :

Il faut n étapes, comme montré dans la preuve. À l'étape p on parcourt la liste restante de longueur $n-p$ pour trouver le minimum.
Le nombre d'opérations sera donc $\sum_{p=0}^{n}(n-p) = \sum_{p=0}^{n} p$ par inversion de l'ordre de sommation.
Donc la complexité est en $\frac{n(n+1)}{2}$, c'est-à-dire en n^2 pour l'ordre de grandeur.
On dit que le tri par sélection est de **complexité quadratique.**
Si je prend une liste deux fois plus longue, il me faudra 4 fois plus de temps pour la trier. Si je prends une liste 10 fois plus longue, il me faudra 100 fois plus de temps. Vous voyez que cela peut devenir problématique lorsque la taille de la liste à trier est très grande (des centaines de milliers, des millions...de données).

B Tri par insertion

Le tri par insertion est le tri naturel du jeu de carte. On parcourt la liste et dès que l'on trouve un élément qui ne suit pas l'ordre correct, on l'insère où il se doit dans la partie déjà triée.

Ainsi, comme dans le tri précédent, à l'étape p, la liste des p premiers éléments est triée. Par contre, la deuxième condition du tri par sélection n'est plus vérifiée. En effet, dans les éléments de la fin de la liste, il faudra en prendre pour les insérer au milieu de la liste déjà triée.
On peut programmer ce tri de plusieurs façon différentes. Voici trois propositions :

- **le naïf :** il est naïf, il est gentil et on l'aime bien. Il a compris ce qu'il fallait faire, mais on pourra pousser la réflexion un peu plus loin pour être plus efficace.
  Pour insérer un élément au milieu de la liste triée, on le fait remonter en échangeant les éléments 2 à 2 jusqu'à trouver la bonne place.

```
def triInsertionNaif(liste):
    # À chaque étape la sous-liste des p premiers éléments est triée.
    for p in range(1,len(liste)):
        # on remonte la liste
        i = p
        while i > 0 and liste[i-1] > liste[i]:
            # dans le mauvais ordre, on fait encore remonter.
            (liste[i],liste[i-1]) = (liste[i-1],liste[i])
            i -= 1
```

Python 7: Le tri par insertion : méthode naïve

- **le pragmatique :** La version naïve est bien, mais cela fait beaucoup d'affectations à chaque fois : dans la réalité, lorsque vous triez votre jeu de carte, vous insérez directement la carte au bon endroit en repoussant toutes les autres d'un cran. Vous n'avez pas besoin de l'insérer dans chaque position intermédiaire. Ici, on se contente donc de garder la carte p en main (on la nomme `element`) et on décale les autres à chaque étape, jusqu'à arriver à la bonne position.

```
def triInsertionPrag(liste):
    for p in range(1,len(liste)):
        #on remonte la liste
        element = liste[p]
        i = p
        while i > 0 and liste[i-1] > element:
            #élément se trouve avant, on repousse d'un cran
            liste[i] = liste[i-1]
            i -= 1
        #on insère entre i-1 et i, c'est-à-dire à la nouvelle position i
        liste[i] = element
```

Python 8: Le tri par insertion : méthode pragmatique

- **le persévérent :** autant suivre la logique du pragmatique jusqu'au bout. Plutôt que de décaler les cartes une par une, on les décale en bloc au moment de l'insertion de la carte p.

```
def triInsertionSlice(liste):
    for p in range(1,len(liste)):
        # on remonte la liste
        i = p
        while i > 0 and liste[i-1] > liste[p]:
            i -= 1
        # on insère entre i-1 et i, c'est-- dire à la nouvelle position i
        liste.insert(i,liste[p])
        liste.pop(p+1)          # attention, il a été décalé par l'insertion
```

Python 9: Le tri par insertion : méthode avec slicing

C Tri à bulles

On continue sur la même lancée. Cette fois-ci on fait remonter les éléments trop gros comme des bulles. Les bulles les plus grosses remontent le plus haut, jusqu'à rencontrer une bulle plus grosse qu'elle.
Concrètement, on fait des parcours de liste en agrippant les bulles une à une. Lorsque l'on fait un parcours sans faire remonter une seule bulle, c'est terminé.

Programmation : On avance dans la liste et on compare les éléments par couple :

- s'ils sont dans le bon ordre, on ne fait rien
- sinon, on les intervertit.

On fait autant de parcours de la liste que nécessaire.

Cet algorithme est un peu plus difficile, vous devez l'avoir compris et être capable d'expliquer, mais on ne peut pas exiger que vous sachiez le programmer en question de cours.

- **Première version :**

```
def triBulles(liste):
    chaos = True                                    #la liste n'est pas triée
    while chaos:
        chaos = False                              # on suppose la liste triée
        for p in range(len(liste)-1):
            if liste[p] > liste[p+1]:
                #on fait remonter la bulle
                (liste[p+1],liste[p]) = (liste[p],liste[p+1])
```

```
                chaos = True    # on n'est pas trié puisqu'on a bougé une bulle
```

Python 10: Le tri à bulles naïf

- **Version améliorée :**
  On s'aperçoit qu'à mesure que l'on réalise des parcours de liste, la fin de la liste est triée. Si la dernière bulle est remontée jusqu'à la position $p+1$ et que l'on a rien modifié derrière, c'est que toute la fin de la liste est triée.
  En plus, la dernière bulle que l'on a fait remonter est plus grosse que toutes celles qui précèdent (sinon, ce serait la bulle plus grosse qui aurait été traitée d'abord). Et cette bulle s'arrête en position $p+1$. Cela veut dire que la sous-liste à partir de $p+1$ est non seulement triée, mais que tous ses éléments sont plus grands que ceux qui précèdent.
  On n'a donc besoin de n'effectuer le tri à bulle que jusqu'à p.
  La condition d'arrêt du tri devient alors lorsque $p = 1$. (s'il ne reste que l'élément 0, il est forcément trié puisqu'il est seul).

```
def triBulles2(liste):
    pmax = len(liste)#la sous liste liste[pmax:] est triée
    while pmax > 1:
        last = 0
        for p in range(pmax-1):
            if liste[p] > liste[p+1]:
                #on fait remonter la bulle
                (liste[p+1],liste[p]) = (liste[p],liste[p+1])
                last = p+1 #où remonte la dernière bulle
        pmax = last
```

Python 11: Le tri à bulles amélioré

Complexité :
Avec la version améliorée, si la liste est déjà triée, alors, il n'y aura qu'un seul parcours de liste.
Par contre, si la liste est triée à l'envers, alors ce sera la situation la plus longue : chaque bulle devra être remontée.
À chaque étape, la longueur de la liste diminue de 1 et on fait un parcours complet.
La complexité est donc $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$: c'est une **complexité quadratique**.

D La complexité

Pour se rendre compte de l'intérêt des questions de complexité, les algorithmes précédents ont été utilisés sur une même liste. À chaque fois, la durée en seconde nécessaire au tri de la liste a été mesuré (grace à la fonction `clock()` de la bibliothèque `time`).
Pour obtenir un élément de comparaison, la liste a également été triée avec le tri par défaut de Python.
Voici les résultats obtenus pour une liste de 1000 éléments (de 0 à 999) rangés aléatoirement, puis pour une liste avec dix fois plus d'éléments.
Une étude expérimentale complète demanderait de faire des statistiques sur un grand nombre de listes, d'essayer avec des listes déjà triées, ou triées en sens inverse... Mais ce n'est pas le but ici.

```
#Pour 1000 éléments
Tri sélection, temps = 0.09130899999991016
Tri insertion naif, temps = 0.12908400000014808
Tri insertion pragmatique, temps = 0.08068600000001425
Tri insertion optimisé, temps = 0.04642300000000432
Tri insertion bulle, temps = 0.43186900000000605
Tri insertion bulle optimisé, temps = 0.23266100000000733
Tri de Python, temps = 0.0002649999998993735

#Pour 10000 éléments
Tri sélection, temps = 6.991663999999901
Tri insertion naif, temps = 12.147036000000071
Tri insertion pragmatique, temps = 7.282917999999881
Tri insertion optimisé, temps = 4.433541000000105
Tri insertion bulle, temps = 43.287115000000085
Tri insertion bulle optimisé, temps = 23.52394400000003
Tri de Python, temps = 0.0033519999999498395
```

Python 12: Temps d'execution des différents tri sur des listes aléatoires

On observe que les écarts de temps sont considérables ! Pour la liste à 10000 éléments (pas très gros), le tri à bulle naïf a mis 43 secondes alors que le tri de Python a mis moins d'un centième de seconde.
Vous voyez donc qu'un algorithme maladroit (même s'il est juste) devient rapidement inutilisable dans la pratique (à moins que vous ne soyez pas pressé...).

Il nous reste beaucoup de travail avant d'obtenir les mêmes performances que ceux qui ont programmé la méthode `sort()` en Python. Mais gardons surtout en mémoire qu'avec un tout petit peu de réflexion, on peut déjà gagner beaucoup de temps. Ainsi entre le tri à bulle naïf et celui optimisé, on a divisé le temps par 2.

On remarque aussi que lorsque l'on a multiplié la taille de la liste par 10, le temps a été globalement multiplié par $10^2 = 100$. CE qui permet de vérifier que la complexité de nos algorithmes est de type **quadratique**[1]. Imaginez donc l'écart de temps pour un tableau de taille 100000. Avec le tri à bulle naïf, on peut imaginer un temps d'environ $4300s \approx 1h10min$ alors que cela demande moins d'une seconde pour la méthode `sort()`.

3 STATISTIQUES D'UNE LISTE DE NOMBRES

A Somme et produit

Sommer ou faire le produit de tous les éléments d'une liste de nombres

```
def somme(liste):
    s = 0
    for i in liste:
        s += i
    return s

def produit(liste):
    p = 1
    for i in liste:
        p *= i
    return p
```

Python 13: Somme des éléments d'une liste

Attention, une somme vide vaut 0, mais un produit vide vaut 1.

B Moyenne

Moyenne des éléments d'une liste (supposée non vide)

```
def moyenne(liste):
    m = 0
    for i in liste:
        m += i
    return m/len(liste) #on suppose len(liste)!= 0
```

Python 14: Moyenne des éléments d'une liste

C Médiane

Médiane des éléments d'une liste (supposée triée). Faites des exemples sur papier pour voir la bonne formule.

```
def mediane(liste):
    # si la liste n'est pas supposée triée, commencer par la trier
```

[1] En fait la méthode Python est en complexité $n \ln n$, mais on ne voit pas bien la différence ici. L'algorithme utilisé est le `Timsort`. C'est une version hybride du tri fusion (qui n'est pas au programme de première année) et du tri par insertion.

```
    lg = len(liste)
    if lg%2 == 0:                              #nombre pair d'éléments
        # moyenne des éléments centraux
        return (liste[lg//2-1]+liste[lg//2])/2
    # sinon renvoie l'élément central
    return liste[lg//2]

#Une autre méthode qui évite la disjonction des cas :
def mediane(liste0):
    lg = len(liste)
    return (liste[lg//2]+liste[(lg-1)//2])/2
```

Python 15: Médiane des éléments d'une liste

4 Recherche dichotomique

Ces algorithmes ne sont pas au programme pour Python, mais ils sont tellement souvent utilisés dans les sujets d'oraux que c'est "comme si".

L'idée est de couper à chaque fois en deux la zone de recherche.

A ⋆ Enjeux du problème : la complexité d'une recherche

Lorsque l'on cherche un élément dans une liste, on parcourt la liste jusqu'à ce qu'on l'ai trouvé.
Au pire des cas, il se trouve à la toute fin et on utilise autant d'opérations que la longueur de la liste : n. On dit que la **complexité au pire** est en $\Theta(n)$.
Mais la plupart du temps, on trouvera l'objet avant. On veut donc savoir, en moyenne, combien de temps il faudra pour trouver l'objet.
Bien sûr cela dépend de la probabilité qu'il a de se trouver à l'une ou l'autre position dans la liste. Ici, on supposera qu'on est sûr qu'il se trouve dans la liste et que la probabilité qu'il se trouve à l'une ou l'autre position dans a liste est la même (répartition équiprobable).
Si on veut connaître la **complexité en moyenne**, on fait donc la moyenne des complexités en fonction de la position de l'objet dans la liste pondérée par la probabilité qu'il s'y trouve.
On note n la longueur de la liste.
Pour chaque indice $k \in [\![0, n-1]\!]$, on note

- $t(k)$ le temps pour trouver l'élément s'il se trouve en position k,
- $P(k)$ la probabilité que l'élément se trouve en position k.

Alors, la complexité en moyenne est donnée par

$$C = \sum_{k=0}^{n-1} t(k)P(k)$$

Or, $\forall k \in [\![0, n-1]\!]$, $P(k) = \frac{1}{n}$ (situation équiprobable)
et $\forall k \in [\![0, n-1]\!]$, $t(k) = k+1$ (temps[2] pour aller à l'indice k).
Ainsi

$$C = \sum_{k=0}^{n-1} t(k)P(k) = \frac{1}{n}\sum_{k=0}^{n-1}(k+1) = \frac{1}{n}\sum_{k=1}^{n} k = \frac{n+1}{2} \approx \frac{n}{2} = \Theta(n)$$

En moyenne, on trouve l'objet avec $\frac{n}{2}$ itération. Cela reste une complexité linéaire comme pour la complexité au pire (si on multiplie la longueur de la liste par 10, alors le temps pour trouver l'objet est aussi multiplié par 10 en moyenne).

En général on ne peut pas faire mieux, car tant que la liste n'a pas été entièrement parcourue, nous n'avons aucune information sur les éléments restants.
Par contre, lorsque la liste est triée, la lecture de chaque élément nous donne des informations sur le reste de la liste. On peut donc utiliser cette information pour réduire le temps de recherche dans la liste.

Nous allons mettre en œuvre cette idée dans deux situations

[2] En fait, comme on raisonne en ordre de grandeur, le $k+1$ pourrait être remplacé par k.

B Recherche dans une liste triée

La recherche **dichotomique** est la recherche naturelle dans une liste triée. Lorsque l'on vous demande de trouver un nombre de 1 à 100 et qu'à chaque essai on vous indique s'il est plus grand ou plus petit, alors la procédure naturelle est de couper l'intervalle en deux à chaque fois : on commence par 50, puis si c'est trop grand, on essaie avec 25...
C'est ce que l'on vous demande de programmer ici.

Principe de l'algorithme :
Pour restreindre l'intervalle de recherche (on divise sa taille par deux environ à chaque itération), on définit les bornes entre lesquelles on cherche l'élément : `mini` (inclus) et `maxi` (exclu[3]).
Au début `mini` vaut le premier indice : 0 et `maxi` vaut le dernier indice +1 : `len(liste)`.
À chaque étape, on prend l'objet du milieu (on note `i` le milieu).

- S'il est égal à `a`, c'est terminé et on renvoie `i`,
- S'il est plus petit, on cherche alors dans [`i`+1,`maxi`[,
- S'il est plus grand, on cherche alors dans [`mini`,`i`[.

Et on continue tant que l'intervalle contient plusieurs éléments. Lorsque l'intervalle est vide, si rien n'a été trouvé, alors c'est que l'objet `a` n'est pas dans la liste et on renvoie une erreur.

Programmation effective :
Dans cette fonction, on restreint peu à peu l'intervalle de recherche . L'intervalle est défini par les bornes.
À chaque itération :

- On prend le milieu de l'intervalle : `i=(maxi+mini)//2`. Par exemple,
  - pour $[1,2,3,4,5,6,7]$, le milieu est 4, et c'est l'élément que l'on prend avec la division entière
$$(\texttt{maxi}+\texttt{mini})//2 = \left\lfloor \frac{\texttt{maxi}+\texttt{mini}}{2} \right\rfloor = \left\lfloor \frac{8+1}{2} \right\rfloor = 4$$
  - pour $[1,2,3,4,5,6,7,8]$ le milieu est à 4,5, on choisit l'élément 5.
$$(\texttt{maxi}+\texttt{mini})//2 = \left\lfloor \frac{\texttt{maxi}+\texttt{mini}}{2} \right\rfloor = \left\lfloor \frac{9+1}{2} \right\rfloor = 5$$
- On compare `a` avec l'élément d'indice `i` :
  - si `a` est égal à l'élément d'indice `i`, c'est terminé : on renvoie `i`,
  - si `a` est strictement supérieur à l'élément d'indice `i`, alors on cherche dans l'intervalle *au delà* de `i`,
  - sinon, `a` est inférieur et on cherche en deça.

On fait les itérations jusqu'à ce que l'on ai trouvé l'élément ou que l'intervalle soit vide (`mini=maxi`).

```
# on suppose la liste de nombres classée par ordre croissant.
def dichotomie(a,liste):
    #intervalle de recherche : [mini,maxi[
    mini = 0
    maxi = len(liste)
    while maxi>mini:
        i = (maxi+mini)//2
        if liste[i] == a:
            return i
        elif liste[i] < a:
            mini = i+1
        else:
            maxi=i #intervalle ouvert à droite
    raise ValueError(str(a)+" n'est pas dans la liste")
```

Python 16: Recherche dichotomique dans une liste classée par ordre croissant

[3] On conserve ainsi la logique de Python : borne inférieur incluse et borne supérieure exclue. Mais ce n'est pas obligatoire.

⋆ **Complexité de la recherche dichotomique :**La complexité au mieux est bien sûr égale à 1 : on tombe sur la bonne valeur au premier essai.
Intéressons-nous à la complexité au pire.
À chaque étape, l'intervalle de recherche est divisé par 2. Au bout de k étapes, la taille de l'intervalle est donc divisée par 2^k.
Pour connaître le nombre d'étapes maximum avant de trouver le bon nombre, il faut chercher le plus petit $k \in \mathbb{N}$, tel que $2^k \geq n$.
On prend donc $k = \log_2(n) = \frac{\ln n}{\ln 2}$
La complexité est en $\Theta(\ln n)$.
C'est une complexité logarithmique. C'est beaucoup mieux qu'une complexité linéaire. Pour s'en convaincre, il suffit de comparer les courbes de $x \mapsto x$ et $x \mapsto \ln x$, lorsque x devient très grand.

C Recherche d'un zéro d'une fonction continue

Remarque : On peut démontrer le théorème des valeurs intermédiaires en utilisant cette démarche.

Le principe est le même. Soit f une application **continue** sur $[a,b]$ avec $f(a)\cdot f(b) < 0$ ($f(a)$ et $f(b)$ de signes contraires et non nuls)

Alors d'après le théorème des valeurs intermédiaires, $\exists c \in]a,b[$ tel que $f(c) = 0$.

Le but est d'obtenir c à ε près.

Pour cela, on découpe à chaque fois l'intervalle en 2 en posant $x = \frac{a+b}{2}$:

- si $f(x) > 0$, c'est que la fonction s'annule après x et on recommence sur l'intervalle $[x,b]$,
- si $f(x) < 0$, c'est que la fonction s'annule avant x et on recommence sur l'intervalle $[a,x]$.

On s'arrête lorsque la taille de l'intervalle est inférieure à $\varepsilon > 0$. On renvoie alors le milieu de l'intervalle pour minimiser l'erreur.

```python
def dichotomie2(f,a,b,epsilon):
    while abs(b-a) > epsilon:
        x = (a+b)/2
        if f(c) == 0:
            return c
        elif f(a)f(c) < 0:
            b = c
        else:
            a = c
    return (a+b)/2

# essai pour obtenir une valeur approchée de racine carrée de 2
def test(x):
    return -(x**2-2)

dichotomie(test,1,2,1e-5)
# renvoie : 1.4142112731933594

test(dichotomie(test,1,2,1e-5))
# renvoie : 6.474772817455232e-06
```

Python 17: Recherche dichotomique du zéro d'une fonction continue