Première Algorithmique	Recherche dichotomique dans un tableau trié	NSI

Objectifs de la séance :

- *Algorithme de recherche dichotomique*
- *Correction de l'algorithme*
- *Notion (informelle) de complexité logarithmique*

Matériel et Logiciel nécessaire :PC

- *un environnement de programmation Python*

Durée estimée : 2 h

A - Introduction

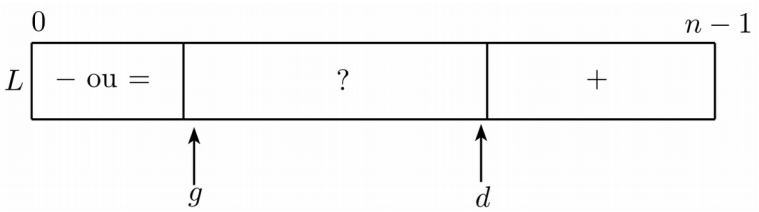
Le fait qu'un tableau soit trié, par exemple par ordre croissant, facilite de nombreuses opérations. L'une d'ente elles est la recherche d'un élément. En effet, il suffit de comparer la valeur recherchée avec la valeur située au milieu du tableau. Si elle est plus petite, on peut restreindre la recherche à la moitié gauche du tableau. Sinon, on la restreint à la moitié droite du tableau. En répétant ce procédé, on divise la zone de recherche par deux à chaque fois. Très rapidement, on parviendra soit à la valeur recherchée, soit à un intervalle devenu vide. On appelle cela la **recherche dichotomique**.

On connaît tous ce principe, pour deviner un nombre en ayant comme réponses à nos tentatives « c'est plus grand » ou « c'est plus petit ». Ce principe est connu en informatique sous le nom de **divier pour régner** et il est appliqué dans de nombreux algorithmes. La recherche dichotomique en est l'expression la plus simple.

B - Description de l'algorithme

On considère une suite L de n éléments classés par ordre croissant. Le problème consiste à établir un algorithme qui permette de dire si un élément x appartient ou pas à la liste L.
L'algorithme recherché ne doit pas effectuer une recherche séquentielle, mais doit utiliser le principe de la dichotomie : à chaque étape de l'itération, l'intervalle de recherche doit être divisé par deux.

1. Supposons que l'algorithme a commencé à travailler pour résoudre le problème, on peut supposer qu'il est arrivé dans la situation suivante :



où $+$ indique la zone des éléments de L qui sont strictement plus grands que x et $-$ ou $=$ indique la zone des éléments inférieurs ou égaux à x.

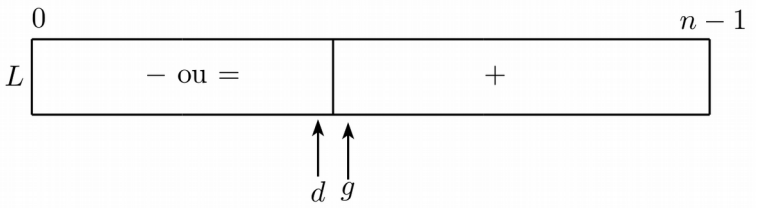
2. La condition d'arrêt serait : $d < g$.
3. Nous allons modifier g et d de manière à diviser par 2 l'amplitude de l'intervalle $[g; d]$ et surtout conserver la situation choisie en **1.**

```
k ← quotient entier de g + d par 2
si L[k] ⩽ x alors
|   g ← k + 1
sinon
|   d ← k − 1
finsi
```

4. Les initialisations $g \leftarrow 0$ et $d \leftarrow n-1$ conviennent.

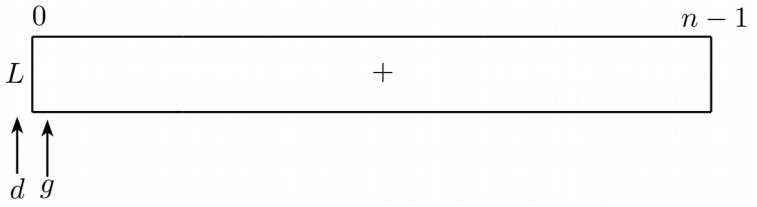
Remarque : À chaque étape, soit g augmente, soit d diminue, la condition d'arrêt sera toujours atteinte. Lorsque la situation finale est atteinte, il suffit d'examiner les diverses possibilités pour conclure :

- **les deux indices ont changés de valeur :**



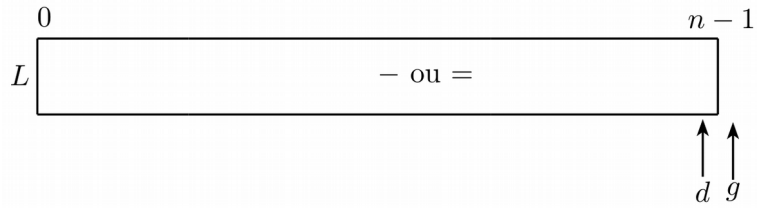
Si x est présent dans L alors son emplacement est d.

- **seul l'indice d a changé de valeur :**



x n'est pas présent dans L.

- **seul l'indice g a changé de valeur**



Si x est présent dans L alors son emplacement est d.

Exercice 1

1. *Tester « à la main » l'algorithme de dichotomie avec la liste* `L = [2, 3, 5, 7, 11, 13, 17, 19]` *et* `x = 5`. Vous utiliserez un tableau pour préciser l'évolution des valeurs des variables au cours de l'exécution de l'algorithme.
2. *Recommencer avec la même liste pour* `x = 2`, *puis pour* `x = 19` *et pour* `x = 6`.

C - Programmation de l'algorithme

Voici le début de la fonction `recherche_dichotomique` avec sa spécification et sa documentation, il reste une partie à écrire en vous inspirant de ce qui a été décrit dans la partie précédente. Vous avec également des … à compléter.

```
def recherche_dichotomique(L : list, x : int or float) -> int:
    """renvoie une position de x dans le tableau L, supposé trié,
    et None si x ne s'y trouve pas"""
    g = ...
    d = ...
    while g <= d:
        # invariant : 0 <= g et d < len(L)
        # invariant : x ne peut se trouver que dans L[g-1..d]
        k = ...
        if L[k] <= x:
            g = k + 1
        else:
            d = k - 1
    # où l'on vérifie si x est dans L
    if ... and L[d] == ...:
        return ...
    else:
        return ...
```

Exercice 2

1. *Programmer cette fonction.*
2. *La construction décrite de l'algorithme dans la partie B permet de donner une preuve de la correction de l'algorithme de recherche dichotomique. Vérifier néanmoins que les invariants indiqués dans le bout de code précédent le sont bien.*
3. *La tester sur la liste de l'exercice 1 et les différentes valeurs proposées.*
4. *Générer par compréhension une liste de 1000 valeurs, la trier par la méthode de votre choix, puis utiliser votre fonction* `recherche_dichotomique` *pour savoir si le nombre* `666` *est présent dans cette liste.*
5. *Combien de valeur sont examinées lors de l'appel à* `recherche_dichotomique([0, 1, 1, 2, 3, 5, 8, 13, 21], 7)` *?*
6. *Donner un exemple d'exécution de* `recherche_dichotomique` *où le nombre de valeurs examinées est exactement quatre.*

D - Efficacité et notion informel de complexité logarithmique

Pour mesurer l'efficacité de la recherche dichotomique, on peut s'intéresser au nombre de valeurs du tableau `L` qui on été examinées pendant l'exécution de `recherche_dichotomique`. C'est exactement le nombre d'itérations de la boucle `while` ou encore le nombre de valeurs prises par la variables `k`, puisque chaque itération de la boucle affecte une valeur différente à la variable `k` et examine l'élément `L[k]`. Pour être plus précis, on peut ajouter le dernier test pour savoir pour avoir le nombre exacte examen de `L[k]`. Le temps d'exécution de `recherche_dichotomique` est directement proportionnel à ce nombre.

Exercice 3

Modifier la fonction `recherche_dichotomique` pour afficher le nombre total de tours de boucle effectués par l'algorithme. Lancer le programme sur des tableaux de tailles différentes : celles indiquées dans le tableau ci-dessous que vous compléterez. On pourra par exemple chercher la valeur `1` dans un tableau ne contenant que des `0`, ce qui correspond au pire cas.

n	k
10	
100	
1 000	
1 000 000	
1 000 000 000	

Plaçons nous dans le pire cas, lorsque la valeur `x` n'apparaît pas dans le tableau L de taille n, ce qui nous oblige à répéter la boucle jusqu'à ce que l'intervalle soit vide.

À la première itération, on va se restreindre à un sous tableau contenant la partie entière de $\frac{n}{2}-1$ ou celle de $\frac{n}{2}$ valeurs, selon le coté choisi. Prenons le cas le moins favorable, avec la partie entière de $\frac{n}{2}$ éléments.

À la seconde itération, on va se retrouver avec au plus la partie entière de $\frac{n}{2^2}$ éléments, puis au plus la partie entière de $\frac{n}{2^3}$ à la suivante et ainsi de suite. On s'arrête lorsque la partie entière du quotient $\frac{n}{2^m}$ vaut `1`. Ainsi, on a $1 \leqslant \frac{n}{2^m} < 2$, où encore $2^m \leqslant n < 2^{m+1}$

Il s'agit donc de déterminer la plus petite valeur de `m` telle que $2^{m+1} > n$. Vérifier que ce sont bien les valeurs données dans le tableau.

On remarque ainsi, que le nombre `m` d'étapes croît très lentement avec la taille `n` du tableau, on dit que cette croissance est logarithmique.