Algorithme de dichotomie sur un tableau trié

L'algorithme de dichotomie est un algorithme de recherche dans un tableau trié. Le fait que le tableau soit trié va permettre à l'algorithme de dichotomie d'être plus efficace que l'algorithme de recherche par parcours séquentiel.

I : On cherche 53 dans un tableau trié où 53 est présent

Voici un exemple sur un tableau de taille 21 trié par ordre croissant : on cherche où se trouve l'élément 53.
L'idée est de regarder au milieu du tableau à chaque fois et, en fonction de ce qu'on observe, éliminer la moitié gauche ou droite du tableau. On recommence alors en regardant au milieu de la moitié restante etc.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
indices																					
éléments	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
Etape 1	?	?	?	?	?	?	?	?	?	?	17	?	?	?	?	?	?	?	?	?	?
Etape 2												?	?	?	?	73	?	?	?	?	?
Etape 3												?	31	?	?						
Etape 4														53	?						

Cet algorithme requiert trois variables :
- g et d donnant la plage $[\![g\ ;d]\!]$ d'indices à garder (ci-dessus $[\![0\ ;20]\!]$ puis $[\![11\ ;20]\!]$ puis $[\![11\ ;14]\!]$puis $[\![13\ ;14]\!]$),
- $m = (g + d)//2$ qui donne l'indice situé au milieu de la plage

II : On cherche 67 dans un tableau trié où 67 n'est pas présent

On cherche l'élément 67 supposé non présent dans le tableau. Remarquer qu'à chaque fois que m<67, c'est g qui se décale à droite de m et qu'à chaque fois que m>67 c'est d qui se décale à gauche de m.

Etape 1	g											m									d
	?	?	?	?	?	?	?	?	?	?	?	17	?	?	?	?	?	?	?	?	?
Etape 2													g				m				d
													?	?	?	?	73	?	?	?	?
Etape 3													g	m		d					
													?	31	?	?					
Etape 4															g,m	d					
															53	?					
Etape 5																g, m,d					
																59					
Etape 6																d	g				

III : Bilan des deux exemples

Il y a donc deux façons de sortir de notre boucle while :
- soit on a trouvé l'élément recherché (m = valeur cherchée)
- soit on ne l'a pas trouvé : on finit par avoir g à droite de d, c'est-à-dire d < g

IV : Implémentation de l'algorithme

```
def dichotomie(tableau, val):

    gauche = 0
    droite = len(tableau) - 1

    while gauche <= droite:

        milieu = (gauche + droite) // 2

        if tableau[milieu] == val:
            return milieu
        elif tableau[milieu] < val:
            gauche = milieu + 1
        else:
            droite = droite - 1
    return -1
```

V : Terminaison de l'algorithme

Pour démontrer que l'algorithme se termine, nous allons utiliser un *variant de boucle* (à ne pas confondre avec un invariant de boucle). Il s'agit d'une quantité à valeurs entières qui doit :
- décroitre strictement à chaque passage dans la boucle (d'où le nom de variant),
- être positive ou nulle pour que la boucle continue.

Si un variant de boucle existe, il va forcément finir par arriver dans le négatif (car il décroit strictement) et donc la boucle ne continuera pas.

`# variant = droite - gauche`

Ici la variant de boucle est `droite-gauche` qui correspond à l'écart entre les deux indices de la zone du tableau qu'il reste à explorer. Cet écart décroit strictement à chaque passage dans la boucle car soit on décale gauche soit on décale droite (soit on a trouvé l'élément cherché et l'algorithme est fini grâce au return).

Par ailleurs la condition de sortie de boucle est bien que droite-gauche < 0 qui équivaut à droite<gauche.

On est donc certain de sortir de la boucle while parce que le variant de boucle devient négatif (ou alors parce qu'on a trouvé l'élément cherché et l'algorithme est fini grâce du return).

VI : Étude expérimentale de la complexité

L'algorithme est tellement efficace que même sur un tableau de taille 100 000 000, python et l'horloge de la machine n'arrivent pas à mesurer le temps d'exécution.

Taille n du tableau	Temps d'exécution (ms)
10 000 000	≈ 0
50 000 000	≈ 0
100 000 000	≈ 0

VII : Démonstration de la complexité logarithmique dans le pire des cas (on note aussi $O(log(n))$)

```
def dichotomie(tableau, val):

    gauche = 0                              # O(1) affectation                          \ O(1)
    droite = len(tableau) - 1               # O(1) affectation + 1 opération            /

    while gauche <= droite:                                                             \
                                                                                        |
        milieu = (gauche + droite)//2       # O(1) affectation + 2 opérations   \       |
                                                                              |     |     |
        if tableau[milieu] == val:          # O(1) test                       |     |     |
            return milieu                   # O(1) return                     > O(1) > O(log(n))
        elif tableau[milieu] < val:         # O(1) test                       |     |     |
            gauche = milieu + 1             # O(1) affectation + 1 opération  |     |     |
        else:                               # O(1)                            |     |     |
            droite = milieu - 1             # O(1) affectation + 1 opération  /     /

    return -1                               # O(1)                                      } O(1)
```

$$O(1) + O(log(n)) + O(1) = O(log(n))$$

Chacune des instructions à l'intérieur de la boucle est en O(1), donc l'intérieur de la boucle est en O(1).
Or il y a environ log_2(n) passages dans la boucle (puisqu'à chaque passage on divise par deux l'écart entre gauche et droite, qui est égal à n au début).
Donc la boucle est en O(1*log(n))=O(1).
Au final la complexité est en O(1)+O(log(n))+O(1) = O(log(n))