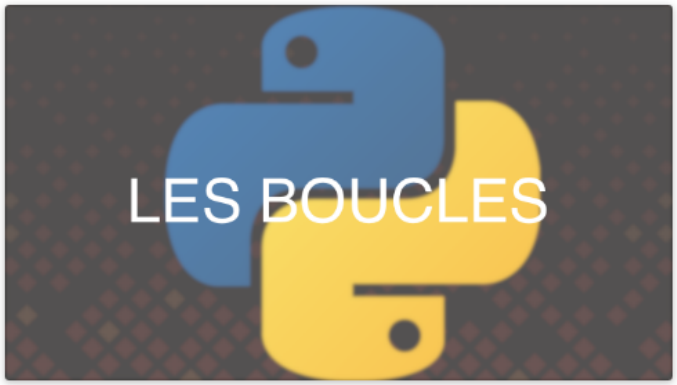
Une nouvelle recherche....

Après la **recherche séquentielle** vue en début de chapitre, puis deux **tris,** nous allons de nouveau chercher un élément dans une liste mais cette fois-ci, TRIÉE ….

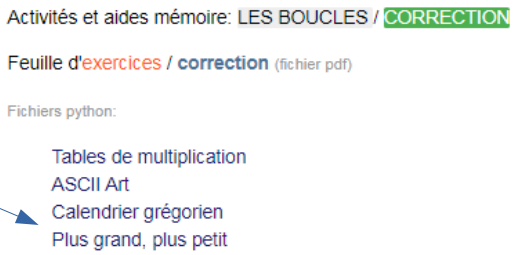


Rappelez vous

On revient sur le JEU du nombre entre 1 et 100

1°) Retrouvez le programme écrit en Python qui cache un nombre entier entre 1 et 100 et demande à l'utilisateur de le retrouver.

Sur le site fredpeuriere.com, il est ici



On comptait même le nombre de coups

Rappelez le nombre maximal de coups.....

et expliquez comment il a été obtenu

.....

….....

Observez et faire fonctionner le fichier *Jeu dicho inversé* depuis le site: décrire son rôle

2°) Nous allons nous intéresser toujours à une liste triée mais, désormais la liste n'est plus forcément une liste de nombres consécutifs.....

a) Comment repérer les éléments de cette liste et surtout comment repérer l'élément du milieu de la liste ? ….....

b) Faire fonctionner le même principe A LA MAIN afin de savoir si la valeur **v = 81** est dans la liste triée suivante : list = [2 , 7 , 19 , 26 , 45 , 70 , 81 , 85 , 86 , 97]

Aide : m indice du milieu, deb l'indice du début de liste, fin celui de fin → lire la VIDEO correspondante **https://drive.google.com/open?id=1F9cf6ms_CX6pNM4Pgu4dFeHaKmGs3jBW**

Essayez de rédiger cela « à la main » : exemple ETAPE *1 – m=int((0+9)/2)=4 , list[4]=45 ≠ 81*

comme list[m] < 81, alors deb = m+1

ETAPE 2 - m=int((5+9)/2)=7 , list[7]=85 ≠ 81 .....

c) Recommencez seul avec v= 77

Fiche d'identité de l' algorithme de **RECHERCHE DICHOTOMIQUE**

Principe : diviser pour régner
La méthode diviser pour régner consiste en trois étapes :

DIVISER : découper le problème initial en sous problèmes ;

RÉGNER : résoudre les sous problèmes

COMBINER : calculer une solution au problème initial à partir des solutions des sous problèmes

Dans un premier temps, la recherche par dichotomie va en fait trouver LA POSITION d'un élément dans la liste triée : il faut comparer l'élément avec la valeur de la case au milieu du tableau ; si les valeurs sont égales, la tâche est accomplie, sinon on recommence dans la moitié du tableau pertinente

Exemple : la recherche d'un mot dans le dictionnaire

Pseudo Code : <u>à compléter</u>

```
v ← élément cherché
deb ← indice du premier élément de la liste
fin ← indice du dernier élément de la liste
m ← ….....................................
Tant que deb <= fin faire
        si v = ….....................  alors
                renvoyer m
        sinon si ….....................  alors
                fin ← m–1
        sinon
                deb ← m+1
        fin si
        m ← ….....................
Fin Tant que
Renvoyer Faux
```

Python <u>à implémenter</u>

TERMINAISON de l'algorithme :
Nous allons nous intéresser au fait que cet algorithme s'arrête....en effet, prouver ici que l'algorithme remplit bien son rôle (CORRECTION de l'algorithme) n'est pas trivial...
On cherche un variant de boucle dans une liste de n éléments
Il s'agit de démontrer que l'algorithme se termine dans tous les cas et qu'il n'y a pas de boucle infinie.
Dans ce cas précis, on peut choisir la suite $\left(\text{fin}_i - \text{deb}_i\right)_{i\in\mathbb{N}}$ des indices de fin et début de la liste considérée.

La condition d'arrêt de la boucle TANT QUE utilisée dans l'algorithme est **Tant que deb<=fin** mais on n'entre pas toujours dans la boucle :

Soit fin_i et deb_i avec i l'indice qui compte la i[ème] itération de la boucle : deb_0 =..... et fin_0 = …...

On note alors $m_i=(deb_i+fin_i)/2$

Si nous sommes à la k[ème] itération, il y a trois grandes possibilités :

- …..... ou ....., l'algorithme se termine, car la boucle n'est pas parcourue.
- $deb_k \leqslant fin_k$ et $v<list[m_k]$, on "entre" dans la boucle : $deb_{k+1}=$..... et $fin_{k+1}=$..... : on a $fin_{k+1}-deb_{k+1}<fin_k-deb_k$
- $deb_k \leqslant fin_k$ et $v>list[m_k]$, on "entre" dans la boucle : $deb_{k+1}=$..... et $fin_{k+1}=$..... On a alors $fin_{k+1}-deb_{k+1}<fin_k-deb_k$

On a donc chaque fois, $fin_{k+1}-deb_{k+1}<fin_k-deb_k$: la suite $(fin_i-deb_i)_{i\in\mathbb{N}}$ est strictement ..... Il existe donc un entier p de $\mathbb{N}$ tel que :

- soit $deb_p>fin_p$, l'algorithme va se …....., car on ne parcourt pas la boucle et l'algorithme renvoie ….....
- soit $v=list[m_p]$ avec $m_p=(deb_p+fin_p)/2$, l'algorithme va se terminer, car on parcourt la boucle et l'algorithme renvoie ….....

L'algorithme se termine donc toujours.

Complexité en temps :

Le logarithme en base 2

DEFINITION : On le note $\log_2$ **et pour** $x\in\mathbb{R}$, $\log_2(2^x)=x$

Lors d'une recherche dichotomique d'un élément v dans une liste triée « list » de longueur n, le nombre *f* de fois dans le cas le plus **défavorable** ($v\notin list$) où la liste sera coupée en deux vérifie $\frac{n}{2^f}=1\Leftrightarrow 2^f=n\Leftrightarrow f=log_2(2^n)$

La complexité en temps dans le pire des cas de l'algorithme de recherche dichotomique est $O(\log_2(n))$ (ce qui est beaucoup mieux que un $O(n^2)$ comme la séquentielle) . Tracez les courbes $x\to\log_2(x)$ et $x\to x^2$ et comparer leurs croissances....

L'algorithme de recherche dichotomique est donc plus efficace que l'algorithme de recherche séquentielle car pour tout x, $x>\log_2(x)$. Néanmoins, il est à noter que la recherche dichotomique utilise un tableau TRIE, ce qui peut aussi avoir un coût en temps...

Exercices

Exercice 1 – *ANALYSER*

Combien de valeurs sont examinées lors d'un appel à la *dichotomie* sur la liste *[0,1,2,2,3,6,9,13,13,20]* pour trouver 7 ? Et avec un appel à une recherche séquentielle ?

Exercice 2 – *TRADUIRE ET DÉVELOPPER*

Écrire en Python une fonction ***Repetition (n)*** qui calcule et renvoie le plus petit entier f tel que $2^f > n$.

Rappeler ce que représente f dans une recherche dichotomique dans une liste de taille n.