THÈME 2 : Algorithme de recherche d'éléments dans un tableau

Peut-on trouver en moins de 20 coups un mot dans un dictionnaire contenant un million de mots ? Nous allons voir que la réponse est oui !

La recherche d'un élément dans un tableau, comme nous l'avons fait durant le thème 1, est une opération en apparence simple et rapide. Toutefois, au prix d'un effort d'abstraction, on peut accélérer la recherche d'un élément et atteindre des niveaux d'efficacité supérieurs.

Cet algorithme est fondamental en informatique.

1) Algorithme naïf

Dans le programme appelé "theme2.py" disponible sur bouillotvincent.github.io, créez une fonction `rechercheNaive` qui recherche si un élément `elt` est contenu dans un tableau d'entiers T, pris comme argument d'entrée.

La fonction `rechercheNaive` renvoie :

- -1 si `elt` n'est pas contenu dans L
- le nombre d'étapes pour trouver l'élément `elt` si `elt` est contenu dans L.

Quelle sera la complexité de cet algorithme dans le pire des cas ? Cela est-il vérifié par le nombre d'étapes indiqué par votre programme ?

2) Algorithme de recherche dichotomique

On donne l'algorithme de recherche dichotomique **dans une liste triée**.

a) À l'aide du papier, d'un crayon et d'un tableau, appliquez cet algorithme en cherchant l'élément 5 sur la liste [1,2,5,9,10,14,17,24,41]. On utilisera les méthodes développées durant le chapitre sur les algorithmes.
Refaire le travail avec l'élément 5 sur la liste [4, 8, 13, 23, 24, 26, 30, 32, 37, 43, 44, 48, 64, 70, 83, 89, 90].

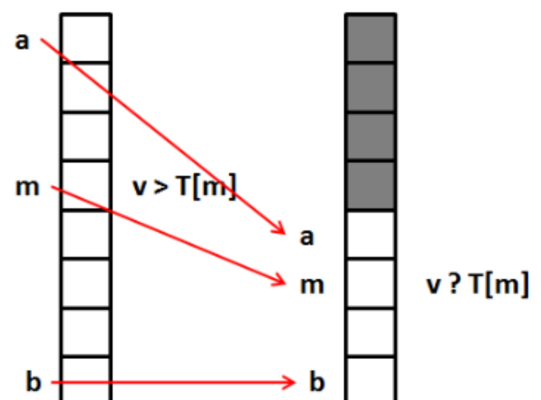
b) Grâce à cet exemple et au schéma ci-contre, sur votre papier, décrire le principe général de fonctionnement de cet algorithme.

c) Dans le programme "theme2.py", traduire l'algorithme en Python en complétant la fonction `rechercheDicho`.

Aide : N'oubliez pas de valider votre fonction en utilisant des tests. On vous donne déjà l'instruction pour vérifier si l'élément 5 est présent dans la liste [1,2,5,9,10,14,17,24,41]

d) Modifier votre fonction afin de :

- renvoyer -1 si l'élément n'appartient pas à la liste ;
- renvoyer le nombre d'étapes pour trouver l'élément si l'élément est contenu dans la liste.



e) Quel est le pire des cas pour cet algorithme ?
À l'aide de tests sur des tableaux de plus en plus grands, essayer de trouver la complexité de cet algorithme dans le pire des cas.

On pourra créer des tableaux efficacement en utilisant des **tableaux en compréhension** :

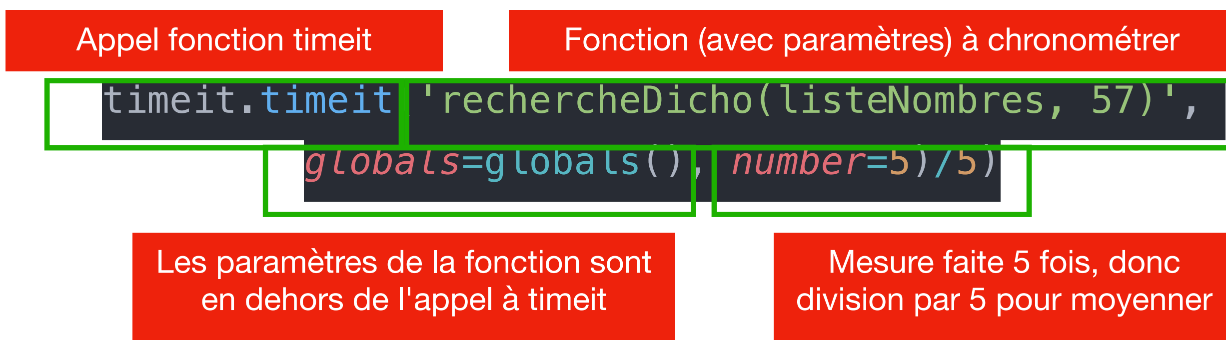
```
T = [ i for i in range(tailleTableau) ]
```

f) Ce deuxième algorithme est-il plus efficace que le premier ?
Avec vos propres mots et en utilisant le schéma ci-contre, expliquez pourquoi.

3) Comparaison d'algorithmes

La librairie timeit permet de faire des mesures de vitesse d'exécution. Pour cela, il faut d'abord l'importer à l'aide de l'instruction `import timeit`.

timeit s'utilise comme suit :



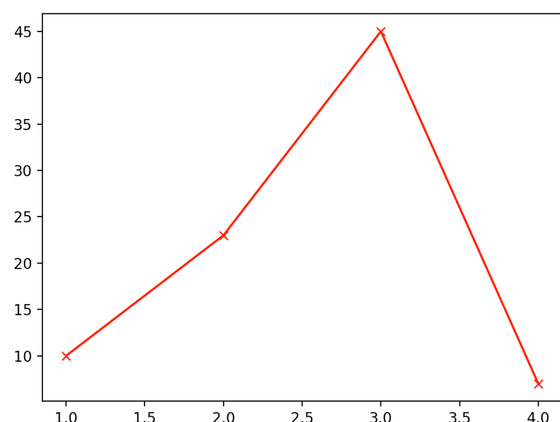
a. Réaliser et afficher quelques mesures de temps d'exécution de votre algorithme naïf ainsi que de votre algorithme de recherche par dichotomie sur des tableaux de taille 100, 1000, 10000.

b. Modifier votre programme de manière à **enregistrer** vos mesures de temps dans deux variables : `tempsNaif` et `tempsDicho` en fonction de tailleTableau.

   Exemple :
   tailleTableau = [10, 100, 1000, 10000]
   tempsNaif = [0,0001 , 0,01, 0,1, 1,2]
   tempsDicho = [0,0001 , 0,001, 0,01, 0,2]

c. Faire **`from matplotlib.pyplot import *.`**
   Réaliser un graphique représentant vos mesures de temps en fonction de tailleTableau. On pourra utiliser une échelle logarithmique en ordonnées.

Un graphique se fait de cette manière :
x = [1,2,3,4]
y = [10, 23, 45, 7]
plot(x, y, 'x-r')
show()



```
VARIABLE
t : tableau d'entiers trié
mil, i_deb, i_fin : nombres entiers
x : nombre entier, entier recherché
tr : booléen

DEBUT
trouve ← FAUX
i_deb ← 0
i_fin ← longueur(t)-1
tant que trouve == FAUX et que i_deb ⩽ i_fin :
    mil ← partie_entière((i_deb+i_fin)/2)
    si t[mil] == x :
         trouve = vrai
    sinon :
         si x > t[mil] :
             i_deb ← mil+1
         sinon :
             i_fin ← mil-1
         fin si
      fin si
fin tant que

renvoyer la valeur de tr
```


```
VARIABLE
t : tableau d'entiers trié
mil, i_deb, i_fin : nombres entiers
x : nombre entier, entier recherché
tr : booléen

DEBUT
trouve ← FAUX
i_deb ← 0
i_fin ← longueur(t)-1
tant que trouve == FAUX et que i_deb ⩽ i_fin :
    mil ← partie_entière((i_deb+i_fin)/2)
    si t[mil] == x :
         trouve = vrai
    sinon :
         si x > t[mil] :
             i_deb ← mil+1
         sinon :
             i_fin ← mil-1
         fin si
      fin si
fin tant que

renvoyer la valeur de tr
```