# Rechercher

INFORMATIQUE COMMUNE - TP n° 1.5 - Olivier Reynet

**À la fin de ce chapitre, je sais :**

- ☞ rechercher un élément dans un tableau séquentiellement ou par dichotomie itérative
- ☞ évaluer le temps d'exécution d'un algorithme avec la bibliothèque time
- ☞ générer un graphique légendé avec la bibliothèque matplotlib

L'objectif de ce TP est d'étudier les algorithmes qui recherchent un élément dans un tableau.

## A Recherche séquentielle

A1. Écrire une fonction de prototype `seq_search(t : list[int], elem : int)-> int` qui impémente l'algorithme de recherche séquentielle d'un élément dans un tableau (cf. algorithme 1). Lorsque l'élément n'est pas présent dans le tableau, la fonction renvoie `None`. Sinon, elle renvoie l'indice de l'élément trouvé dans le tableau. Vérifier que cet algorithme fonctionne sur un tableau d'entiers de 20 éléments rempli aléatoirement.

**Solution :**

```
def seq_search(t, elem):
    for i in range(len(t)):
        if t[i] == elem:
            return i
    return None
```

**Algorithme 1** Recherche séquentielle d'un élément dans un tableau

```
1 : Fonction RECHERCHE_SÉQUENTIELLE(t, elem)
2 :     n ← taille(t)
3 :     pour i de 0 à n − 1 répéter
4 :         si t[i]= elem alors
5 :             renvoyer i                    ▷ élément trouvé, on renvoie sa position dans t
6 :     renvoyer l'élément n'a pas été trouvé
```

A2. Dans le pire des cas, combien d'opérations élémentaires seront nécessaires pour rechercher séquentiellement un élément dans un tableau de taille $n$ ?

**Solution :** Soit $k$ un entier naturel. Dans le pire des cas, c'est-à-dire lorsque l'élément n'est pas présent dans le tableau, il faudra parcourir tout le tableau. $C(n) = kn$ opérations élémentaires pour réaliser la recherche séquentielle : le coût est proportionnel à la taille du tableau.
Dans le meilleur des cas, l'élément cherché est le premier du tableau. On a alors $C(n) = k = O(1)$, un nombre d'opérations qui ne dépend pas de $n$.
Il faut noter que cet algorithme est effectué lorsqu'on écrit `x in t` pour savoir si `x` est un élément de la liste `t`. Cette opération n'a donc pas un coût anodin.

## B Recherche dichotomique

On suppose maintenant que le tableau dans lequel la recherche doit être effectuée est **trié**.

B1. Écrire une fonction de signature `trier_insertion(L: list[int])` qui implémente le tri par insertion ascendant d'une liste d'entier. Vérifier l'algorithme sur une liste de 20 entiers aléatoirement choisis.

**Solution :** cf. cours ;-)

B2. Écrire une fonction de prototype `dichotomic_search(t : list[int], elem : int)-> int` qui implémente l'algorithme de recherche d'un élément par dichotomie (cf. algorithme 2). Lorsque l'élément n'est pas présent dans le tableau, la fonction renvoie `None`. Sinon, elle renvoie l'**indice** de l'élément trouvé dans le tableau. Vérifier que cet algorithme fonctionne sur un tableau d'entiers de 20 éléments rempli aléatoirement et trié.

**Algorithme 2** Recherche d'un élément par dichotomie dans un tableau **trié**

```
1 : Fonction RECHERCHE_DICHOTOMIQUE(t, elem)
2 :     n ← taille(t)
3 :     g ← 0
4 :     d ← n-1
5 :     tant que g ⩽ d répéter                ▷ ⩽ cas où valeur au début, au milieu ou à la fin
6 :         m ← (g+d)//2                      ▷ Division entière : un indice est un entier!
7 :         si t[m] = elem alors              ▷ avoir de la chance n'est pas exclu!
8 :             renvoyer m                    ▷ l'élément a été trouvé
9 :         sinon si t[m] < elem alors
10 :            g ← m + 1                     ▷ l'élément devrait se trouver dans t[m+1, d]
11 :        sinon
12 :            d ← m - 1                     ▷ l'élément devrait se trouver dans t[g, m-1]
13 :    renvoyer l'élément n'a pas été trouvé
```

B3. On suppose que la longueur du tableau est une puissance de 2, c'est à dire $n = 2^p$ avec $p \geqslant 1$. Combien d'itérations la boucle `tant que` de l'algorithme 2 comporte-t-elle ? En déduire le nombre d'opérations élémentaires effectuées dans le cas où l'élément est absent (c'est-à-dire le pire des cas), en fonction de $n$. Comparer avec l'algorithme de recherche séquentielle.

**Solution :** Supposons que la taille de la liste soit une puissance de 2 : $n = 2^p$. Soit k, le nombre d'itérations nécessaires pour terminer l'algorithme. Lors de la dernière itération, le tableau considéré par les indices `g` et `d` ne contient plus qu'un seul élément.
Comme on divise par deux la taille du tableau à chaque tour de boucle, à la fin de l'algorithme, si on a effectué $k$ itérations, on a nécessairement :

$$1 = \frac{n}{2^k} = \frac{2^p}{2^k}$$

On en déduit que $k = \log_2 n$. Le nombre d'itérations est donc $\log_2 n$ et le nombre d'opérations élémentaires exécutées proportionnel à $\log_2 n$. On dit que cet algorithme est de complexité logarithmique en $O(\log n)$ (cf. cours du deuxième semestre), ce qui est nettement plus efficace que l'algorithme de recherche séquentielle qui est de complexité linéaire en $O(n)$.
Néanmoins, il faudra trier le tableau en amont pour pouvoir chercher avec cette méthode, ce qui nécessite également un certain nombre d'opérations élémentaires.

B4. La recherche dichotomique fonctionne-t-elle sur les listes non triées ? Donner un contre-exemple si ce n'est pas le cas.

**Solution :** On peut prendre par exemple la recherche de l'élément `11` dans le tableau `[1,10,0,44,11,19,37]`
Lors de la première itération, l'indice `m` vaut 3. On va donc comparer 11 à 44 et lancer la recherche à gauche de 44. On en conclura que l'élément n'est pas présent dans le tableau.

B5. Soit `t` un tableau de chaînes caractères trié dans l'ordre lexicographique. Peut-on utiliser la recherche dichotomique programmée ci-dessus pour rechercher une chaîne de caractère ? Pourquoi ? On pourra prendre par exemple le tableau `['', 'A', 'ACCTA', 'ACGT', 'AT', 'CACG', 'CTCACGA', 'GGTCA', 'GTCAAA', 'TAGCTGA', 'TT']`.

## C Rechercher dans une liste imbriquée et jouer avec...

C1. Écrire une fonction de signature `empty_nested_lists(n:int)-> list[int]` qui renvoie une liste de n listes vides.

**Solution :**

```
def empty_nested_lists(n):
    L = []
    for i in range(n):
        L.append([])
    return L
    #return [[] for _ in range(n)]
```

C2. Écrire une fonction de signature `alea_nested_list(n:int, h:int)-> list[list[int]]` qui renvoie une liste de n listes d'entiers choisis aléatoirement en 0 et 100 exclu et dont la taille des sous-

listes est aléatoire mais ne dépasse jamais h. Par exemple, un résultat possible de `alea_nested_list(5, 4)` est `[[2], [61, 88, 64, 86], [73, 2, 50], [], [53, 94]]`.

**Solution :**

```
def alea_nested_list(n, h):
    L = []
    for i in range(n):
        SL = []
        for j in range(randrange(h+1)):
            SL.append(randrange(100))
        L.append(SL)
    return L
    #return [[randrange(100) for _ in range(h+1)] for _ in range(n)]
```

C3. Écrire une fonction de prototype `flatten(L : list[list[int]])-> list[int]` qui renvoie la liste mise à plat. Par exemple, pour la liste `[[39, 89], [], [51, 24, 84, 27], [], [39, 44]]` cette fonction renvoie `[39, 89, 51, 24, 84, 27, 39, 44]`.

**Solution :**

```
def flatten(L : list[list[int]]) -> list[int] :
    F = []
    for i in range(len(L)):
        for j in range(len(L[i])):
            F.append(L[i][j])
    return F
```

C4. Écrire une fonction de prototype `nested_sum(L : list[list[int]])-> int` qui renvoie la somme des éléments des sous-listes d'une liste imbriquée.

**Solution :**

```
def nested_sum(L : list[list[int]]) -> int :
    s = 0
    for i in range(len(L)):
        for j in range(len(L[i])):
            s += L[i][j]
    return s
```

C5. Écrire une fonction de prototype `sublists_sizes(L : list[list[int]])-> list[int]` qui renvoie la liste des tailles des sous-listes de la liste imbriquée. Par exemple, pour la liste `[[39, 89], [], [51, 24, 84, 27], [], [39, 44]]` cette fonction renvoie `[2, 0, 4, 0, 2]`.

**Solution :**

```
def sublists_sizes(L: list[list[int]]) -> list[int]:
    sizes = []
```

```
        for i in range(len(L)):
            sizes.append(len(L[i]))
        return sizes
```

C6. Appliquer la recherche dichomotique à une liste imbriquée en la mettant à plat et en la triant.

**Solution :**

```
    def search_nested_list(L: list[list[int]], elem) -> int:
        F = flatten(L)
        trier_insertion(F)
        return dichotomic_search(F, elem)
```