**TP : algorithmes de recherche**

# 1 Recherche dans un tableau

On considère une liste nommée `t` de nombres (ou de chaînes de caractères). On voudrait écrire un algorithme qui détermine si une valeur donnée `x` est un élément de cette liste.

## 1.1 Recherche séquentielle

On regarde si le premier élément de la liste est égal à `x`, sinon, on passe au deuxième...

1. Écrire une fonction booléenne `est_dansTableau` qui prend en argument un entier $x$ et un tableau $t$ d'entiers et teste l'appartenance de $x$ à $t$.
2. Modifier votre fonction de façon à ce qu'elle renvoie le premier indice où la valeur $x$ apparaît dans le tableau. On renverra `None` si $x$ est absent du tableau. On pourra appeler `indice` cette nouvelle fonction.
3. Donner dans le pire des cas, le nombre de comparaisons. Comment qualifieriez-vous la complexité temporelle de cette fonction ?

## 1.2 Recherche dichotomique pour un tableau trié

On suppose cette fois-ci que l'on dispose d'un tableau trié. Nous allons voir dans ce cas une méthode de recherche beaucoup plus efficace.

Il y a une situation concrète, dans laquelle nous sommes emmenés à chercher un mot dans une liste triée : la recherche d'un mot dans un dictionnaire. Il est clair que nous ne cherchons pas le mot comme précédemment. Nous pouvons utiliser une stratégie dite de dichotomie, «l'art de diviser en deux».

Prenons par exemple $t = [2, 4, 7, 12, 15, 20, 25, 28, 31, 36]$ avec $x = 25$.

On prend l'élément médian (ou juste inférieur) du tableau, c'est-à-dire 15 . Notre nombre $x = 25$ est supérieur à 15, donc si $x$ est présent, il se trouve dans le sous-tableau à droite de 15, c'est-à-dire dans $[20, 25, 28, 31, 36]$. On recommence, 28 est le milieu du sous-tableau, mais $25 < 28$ donc 25 s'il est présent est dans le sous-tableau à gauche de 28, c'est-à-dire dans $[20, 25]$ . On continue ce procédé, jusqu'à ce que l'on tombe sur 25, ou jusqu'à ce que l'on tombe sur un tableau vide, auquel cas, la valeur $x$ est absente du tableau.

4. Écrire en pseudo-code (en français) une fonction `recherche_dichotomie` qui prend en argument un nombre $x$ et un tableau $t$ et et renvoie vrai si l'élément $x$ appartient à $t$. On pourra démarrer ainsi :

```
Algo: recherche_dichotomie
Données: x un nombre et t une liste triée
Résultat: vrai si x est un élément de t et faux sinon
n = longueur de t
a = 0  # indice de la borne  gauche du tableau
b = n-1 # indice de la borne droite
...
```

5. Implémenter votre fonction en Python et tester là avec le tableau précédent et les valeurs $x = 25$ et $x = 24$.

On souhaite maintenant prouver la terminaison et déterminer la complexité de notre algorithme. On note $n$ la longueur du tableau $t$, puis $a_k$ et $b_k$ les valeurs respectives des variables $a$ et $b$ à la sortie de la $k$-ième itération pour $k \in \mathbb{N}^*$. On pose aussi $a_0 = 0$ et $b_0 = n - 1$, qui correspondent aux valeurs des variables $a$ et $b$ avant la première itération.

6. Est-il vrai qu'à chaque itération la longueur $b - a$ est divisée par deux ? Au moins par deux ? Au plus par deux ? On pourra démontrer que :

$$b_{k+1} - a_{k+1} + 1 \leqslant \frac{b_k - a_k + 1}{2}.$$

7. En déduire que l'algorithme se termine.
8. On note $p$ le nombre d'itérations, démontrer que $p \leqslant \log_2(n)+1$. Comment qualifie-t-on cette complexité ?
9. Démontrer que l'algorithme est faux si l'on remplace la condition «Tant que $b-a \geqslant 0$» par «Tant que $b-a>0$» en prenant le tableau ci-dessus et la valeur $x=25$. Comment modifier la fin de l'algorithme pour qu'il soit juste, si l'on garde la condition «Tant que $b-a>0$» ?

# 2 Comment trier une liste ?

C'est une problématique très importante en informatique. Il existe de nombreux algorithmes de tri, certains livres y sont même entièrement consacrés. Il existe dèjà en Python, deux instructions prêtes à l'emploi qui permettent de trier des tableaux :

- la fonction `sorted` qui renvoie un nouveau tableau trié (le tableau pris en argument n'est pas modifié).

```
>>> t = [5,9,8,12,3]
>>> sorted(t)
[3, 5, 8, 9, 12]
>>> t # le tableau de départ n'est pas modifié
[5, 9, 8, 12, 3]
```

- la méthode `sort` qui modifie le tableau et le trie.

```
 >>> t.sort()
>>> t
[3, 5, 8, 9, 12] # le tableau de départ a été modifié
```

  Cette méthode présente l'avantage d'être économique en mémoire car on n'a pas créé de tableau supplémentaire, mais il faut avoir conscience que l'on a perdu définitivement l'ordre initial des éléments du tableau.

Nous proposons dans ce TP de découvrir deux algorithmes de tri.

## 2.1 Le tri à bulles

Voici le principe : on «balaye» le tableau une première fois, et on échange deux éléments consécutifs du tableau s'ils ne sont pas dans le bon ordre. Après ce «balayage», le dernier élément du tableau est le plus grand (résultat à prouver). On recommence ensuite à «balayer» mais avec le tableau privé du dernier élément car il est déjà à la bonne place. Si au cours d'un «balayage», aucun échange n'est effectué, c'est que le tableau est trié et on s'arrête.

10. Mettre en oeuvre sur papier, cette méthode avec le tableau $t=[5,2,3,1,4]$.
11. Écrire une procédure `triBulle` qui prend en argument un tableau $t$ et le trie. En particulier, cette procédure ne renvoie rien (avoir conscience que le tableau initial est perdu).
12. Déterminer dans le «meilleur des cas» et dans le «pire des cas», le nombre de comparaisons que l'on effectue, en fonction de la taille $n$ du tableau pris en argument.
13. Démontrer qu'après un «balayage», le plus grand terme se trouve effectivement en dernière position.

## 2.2 Le tri par sélection du minimum

Voici le principe : on «sélectionne» le plus petit élément du tableau et on le permute avec $t[0]$ le premier élément du tableau. On recommence ensuite avec $t[1:n]$ le sous-tableau privé du premier élément, on sélectionne le plus petit élément de $t[1:n]$ et on le permute avec son premier élément $t[1]$. On recommence jusqu'à ce que notre sous-tableau n'ait plus que deux éléments.

14. Mettre en oeuvre sur papier, cette méthode avec le tableau $t = [5, 2, 3, 1, 4]$.
15. Écrire le pseudo-code de cet algorithme en utilisant notamment l'instruction «échanger les valeurs ...».
16. Écrire une procédure `triSelection` qui prend en argument un tableau $t$ et le trie. En particulier, cette procédure ne renvoie rien (avoir conscience que le tableau initial est perdu).
17. Déterminer le nombre de comparaisons que l'on effectue, en fonction de la taille $n$ du tableau pris en argument.

Ces deux tris ne sont en fait pas très performants, vous verrez l'année prochaine les «tri fusion» et «quick-sort» qui sont beaucoup plus efficaces.