CORRIGÉ du 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.

```
def est_dansTableau(x,t):
    """Fonction booléenne qui teste si x est un élément du tableau t
    """
    n = len(t)
    for k in range(n):
        if t[k] == x:
            return True
    return False # arrivé ici, on est sûr que x est absent
```

   La même version en plus «pythonesque» :

```
def est_dansTableau(x,t):
    """Fonction booléenne qui teste si x est un élément du tableau t
    """
    n = len(t)
    for element in t: # on itère directement sur le tableau:
        if element == x:
            return True
    return False # arrivé ici, on est sûr que x est absent
```

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.

```
def indice(x,t):
    """Fonction qui renvoie le plus petit indice de x dans le tableau t si x est dans t
    """
    n = len(t)
    for k in range(n):
        if t[k] == x:
            return k
    return None # arrivé ici, on est sûr que x est absent
```

3. Donner dans le pire des cas, le nombre de comparaisons que l'on effectue. Comment qualifieriez-vous la complexité temporelle de cette fonction ?

   Si on a beaucoup de chances, x est le premier élément du tableau et une seule comparaison sera nécessaire. Dans le pire des cas, x n'est pas dans le tableau, on devra alors réaliser toutes les boucles, c'est-à-dire n comparaisons si n est la longueur de la liste. On dit alors que dans **le pire des cas**, la complexité temporelle est en $O(n)$, donc linéaire.

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 une valeur et t une liste triée et
Résultat: vrai si x est un élément de t et faux sinon
n = longueur de t
a = 0
b = n-1
Tant que b-a >= 0  # tant qu'il y a un élément dans [a,b]
    m = quotient de a+b par 2 # milieu de a et b
    si t[m] = x
       renvoie vrai
    sinon si x < t[m]
       b = m - 1
    sinon
       a = m + 1
# arrivé ici, ici b-a < 0 donc il n'y a plus de mots d'indice dans [a,b], le mot est donc absent
Renvoie faux
```

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

```
def recherche_dicho(x,t):
    """Données: t est une liste triée de nombres ou de chaînes de caractères
                x est un nombre ou une chaîne de caractère
       Résultat: si x est un élément de liste, renvoie l'indice de x dans t,
       sinon ne renvoie rien "
       Traitement: on utilise une méthode de dichotomie
    """
    a=0
    b=len(t)-1
    while b-a >= 0:   # tant qu'il y a un élement d'indice dans [a,b]
        m = (a+b)//2  # indice médian
        if x == t[m]:
            return(m)
        elif x < t[m]:
            b = m - 1
        else:
            a = m + 1
    # arrivé ici b-a < 0 donc il n'y a plus de mots d'indice dans [a,b], le mot est donc absent
    return(None)
```

Donnons les traces avec $x = 25$ et $x = 24$ et $t = [2, 4, 7, 12, 15, 20, 25, 28, 31, 36]$:

a	0	5	5	6
b	9	9	6	6
$b-a \geqslant 0$	V	V	V	V
m	4	7	5	6
$t[m]$	15	28	20	25
$t[m]=25$	F	F	F	V

a	0	5	5	6	6
b	9	9	6	6	5
$b-a \geqslant 0$	V	V	V	V	F
m	4	7	5	6	
$t[m]$	15	28	20	25	
$t[m]=24$	F	F	F	F	

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 :

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

En observant les traces précédentes, il semble que $b-a$ soit au moins divisé par 2 à chaque itération. Prouvons le :

Comme l'entier m est la partie entière de $\frac{a+b}{2}$ le milieu de $[a,b]$, il y a deux cas :

- si m est le milieu de $[a,b]$, alors $b_{k+1} - a_{k+1} = \frac{b_k - a_k}{2} - 1$.
- si m n'est pas le milieu de $[a,b]$. Alors $m = \frac{a+b}{2} - \frac{1}{2}$. Il y a alors deux sous-cas :
  Si $a_{k+1} = m+1$ et $b_{k+1} = b_k$, alors $b_{k+1} - a_{k+1} = \frac{b_k - a_k}{2} - \frac{1}{2}$.
  Si $a_{k+1} = a_k$ et $b_{k+1} = m-1$ alors $b_{k+1} - a_{k+1} = \frac{b_k - a_k}{2} - \frac{1}{2}$.

Dans tous les cas, on a $b_{k+1} - a_{k+1} \leqslant \frac{b_k - a_k}{2} - \frac{1}{2}$, ce qui se réécrit

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

7. En déduire que l'algorithme se termine.

Si l'algorithme boucle indéfiniment, alors la condition $b-a \geqslant 0$ est toujours vraie, donc on a pour tout entier k, $b_k - a_k + 1 \geqslant 1$. Mais alors par récurrence immédiate,

$$\forall k \in \mathbb{N},\ b_k - a_k + 1 \leqslant \frac{b_0 - a_0 + 1}{2^k} = \frac{n}{2^k}.$$

Or $\frac{n}{2^k} < 1 \iff 2^k > n \iff k > \log_2(n)$. Cela montre que pour $k > \log_2(n)$, on a $b_k - a_k + 1 < 1$, contradiction. L'algorithme se termine et le nombre d'itérations p est inférieur ou égal au plus petit entier supérieur à $\log_2(n)$. Donc $p \leqslant \log_2(n) + 1$. En particulier la complexité est logarithmique.

8. On note p le nombre d'itérations, démontrer que $p \leqslant \log_2(n)+1$. Comment qualifie-t-on cette complexité ?

Nous avons donc prouvé que dans le pire des cas, le nombre d'itérations est majoré par $\lceil \log_2 n \rceil + 1$. C'est une **complexité logarithmique**. C'est incroyablement mieux qu'une complexité linaire.

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 $b-a > 0$» ?

Bien que l'algorithme ne soit pas très difficile à écrire, il faut être très vigilant à la condition d'arrêt. Quelques pièges se présentent : par exemple si l'on avait écrit «Tant que $b-a > 0$» au lieu de «$b-a \geqslant 0$», en reprenant la trace de l'exemple précédent avec $v = 25$, on aurait eu

a	0	5	5	6
b	9	9	6	6
$b-a > 0$	V	V	V	F
m	4	7	5	
$t[m]$	15	28	20	
$t[m]=25$	F	F	F	

et l'algorithme aurait renvoyé Faux alors que 25 est bien présent dans le tableau.

Lorsque l'algorithme s'arrête de boucler, on a alors $b - a \leqslant 0$ et donc on peut avoir $a = b$, il faut donc procéder à un dernier test pour savoir si x est égal à a.

Remarques :

- la dichotomie n'est pas une méthode gadget, nous la retrouverons par exemple pour la résolution d'équations algébriques. C'est une stratégie de type «diviser pour régner».
- Dans cet algorithme, on maintient **l'invariant** (de boucle) suivant : les valeurs du tableau d'indice strictement inférieur à a sont strictement inférieures à x et les valeurs du tableau d'indice strictement supérieur à b sont strictement supérieur à x, ce qui peut se schématiser par le tableau suivant :

indice		a		b	
valeur	$<$		v		$>$

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 $t = [5, 2, 3, 1, 4]$.

    Voici l'évolution des valeurs de t après chaque échange de valeurs notées en gras :

    Balayage n°1 [2, 3, **1, 5**, 4] Balayage n°2

    [**2, 5**, 3, 1, 4]

    [2, **3, 5**, 1, 4] [2, 3, 1, **4, 5**] [2, **1, 3,** 4, 5]

Balayage n°3 [**1, 2**, 3, 4, 5]

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).

    Voici une première version, où l'on balaye toujours $n-1$ fois le tableau même si au cours d'un balayage, il n'y a pas eu d'échanges.

```
def triBulle(t):
    """Données: t un tableau de nombres
       Résultat: trie le tableau t"""
    n =len(t)
    for i in range(n-1): # on balaye n-1 fois
        for k in range(0, n-1-i):
            if t[k+1] < t[k]: # dans ce cas on permute t[k+1] et t[k]
                temp = t[k]
                t[k] = t[k+1]
                t[k+1] = temp
```

    Voici une deuxième version améliorée.

```
def triBulle(t):
    """Données: t un tableau de nombres
       Résultat: trie le tableau t"""
    n =len(t)
    i = 0
    des_echanges = True
    while i < n-1 and des_echanges:  # on balaye au pire n-1 fois, tant qu'il y a des échanges
        des_echanges = False
        for k in range(0, n-1-i):
            if t[k+1] < t[k]: # dans ce cas on permute t[k+1] et t[k]
                temp = t[k]
                t[k] = t[k+1]
                t[k+1] = temp
                des_echanges = True
        i += 1
```

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.

    Dans le meilleur des cas, le tableau est trié, et dans ce cas, après le premier balayage, aucun échange n'a été effectué et donc l'agorithme s'arrête. Au cours de ce balayage, seulement n comparaisons ont été effectuées.

    Dans le pire des cas, on effectue toutes les itérations. Au cours du balyage $n°i$, on effectue $n-1-i$ comparaisons, le nombre total de comparaisons est donc

$$\sum_{i=0}^{n-2}(n-1-i) = (n-1)+(n-2)+\cdots+1 = \frac{n(n-1)}{2} = O(n^2).$$

    Conclusion : la complexité est linéaire dans le meilleur des cas, mais quadratique dans le pire des cas.

    Remarque : on peut prouver qu'en «moyenne» sa complexité est encore quadratique.

13. Démontrer qu'après un «balayage», le plus grand terme se trouve effectivement en dernière position.

    Prouvons cette propriété par récurrence sur n la taille du tableau. Si $n=1$, il n'y a qu'un seul élément dans le tableau, donc ok. Supposons que c'est vrai au rang n, et considérons t un tableau de longueur $n+1$. On compare les deux premiers éléments, et on les échange s'ils ne sont pas dans le bon ordre, on a donc $t[0] \leqslant t[1]$. On termine ensuite le balayage. Le sous-tableau $t[1:n]$ a donc été balayé, et il est de longueur n, donc par hypothèse de récurrence, son dernier terme est le plus grand. En particulier, ce dernier terme est plus grand que $t[1]$ et donc aussi que $t[0]$. Ceci montre que $t[n]$ est le plus grand terme du tableau t après un balayage.

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 $t = [5, 2, 3, 1, 4]$.

    Le minimum de t est 1, on échange 1 et 5 donc $t = [\mathbf{1}, 2, 3, \mathbf{5}, 4]$.

    Le minimum de $t[1:5] = [2, 3, 5, 4]$ est 2, donc pas d'échange : $t = [1, \mathbf{2}, 3, 5, 4]$

    Le minimum de $t[2:5] = [3, 5, 4]$ est 3, donc pas d'échange : $t = [1, 2, \mathbf{3}, 5, 4]$

    Le minimum de $t[3:5] = [5, 4]$ est 4, donc on échange 4 et 5 donc $t = [1, 2, 3, \mathbf{4}, \mathbf{5}]$

15. Écrire le pseudo-code de cet algorithme en utilisant notamment l'instruction «échanger les valeurs ...».

```
n <- longueur de t
Pour i de 0 à n-2
    i0 <- indice du minimum du sous-tableau t[i:n]
    échager dans t les valeurs t[i] et t[i0+i] # cf remarque ci-dessous
Fin Pour
```

    Remarque : attention, si x est le maximum du sous-tableau $t[i:n]$, et que son indice dans $t[i:n]$ vaut i_0, alors son indice dans t vaut $i_0 + i$.

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).

    On va créer une sous-fonction qui renvoie l'indice du minimum d'un tableau :

```
def indice_du_minimum(t):
    """renvoie l'indice d'un élément minimal du tableau t"""
    n = len(t)
    mini = t[0]
    indice_de_mini = 0
    for i in range(1,n):
        if t[i] < mini:
            mini = t[i]
            indice_de_mini = i
    return indice_de_mini
```

    On termine :

```
def triSelection(t):
    n = len(t)
    for i in range(n-1): # de i = 0  à n-2
        i0 = indice_du_minimum(t[i:n]) # attention son indice dans t vaut i0+i
        temp = t[i]
        t[i] = t[i0+i]
        t[i0+i] = temp
```

17. Déterminer le nombre de comparaisons que l'on effectue, en fonction de la taille n du tableau pris en argument.

    Dans tous les cas, l'algorithme effectue le même nombre d'opérations. Si t est un tableau de longueur p, la fonction `indice_du_minimum` réalise $p - 1$ comparaisons. Pour chaque valeur de i, la fonction `indice_du_minimum` s'applique au tableau $t[i:n]$ qui est de longueur $n - i$, elle réalise donc $n - i - 1$ comparaisons. Le nombre total de comparaisons est donc de

$$\sum_{i=0}^{n-2}(n-1-i) = (n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2} = O(n^2).$$

À nouveau, on obtient une complexité quadratique et dans tous les cas. Du point de vue du nombre de comparaisons, cet algorithme est donc moins performant que le tri à bulles.

Remarque : nous avons mesuré la complexité en terme de nombre de comparaisons, mais on aurait pu aussi s'intéresser au nombre d'échanges...

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, de complexité quasi-linéaire, en $O(n \ln n)$.