TP7

Partie 1 — Recherche

Ex. 1

Recherche dans un tableau non trié

On veut chercher une valeur dans un tableau (également appelé une liste en Python) qui contient une suite d'entiers non triés.

Dans ce cas, il n'y a pas d'algorithme plus efficace que de parcourir le tableau jusqu'à trouver la valeur cherchée, puisqu'elle peut se trouver dans n'importe quelle position. Cet algorithme a une complexité O(n), ou n est la longeur du tableau, puisque dans le pire des cas il faut en parcourir la totalité pour vérifier que la valeur cherchée n'y apparait pas.

On appelle ce type de recherche une « recherche linéaire ».

Question 1 : Écrire le code de la fonction `linear_search(x, a)`. La fonction doit retourner un entier (entre `0` et `len(a) - 1`) qui correspond à la position de l'élément `x` dans le tableau `a` (ou de sa première occurrence, si `x` apparait plusieurs fois dans `a`), ou bien l'entier `-1` si `x` n'apparait pas dans `a`. Utiliser l'algorithme de recherche linéaire pour cet exercice.

Question 2 : Pour tester la fonction `linear_search`, afficher les résultats de la recherche des valeurs `2`, `5`, `10` et `6` dans le tableau `a1`.

Recherche dans un tableau trié

Si le tableau où on veut chercher la valeur est déjà trié, on peut utiliser un algorithme plus efficace que la recherche linéaire, c'est-à-dire la recherche dichotomique.

On commence par chercher la valeur dans tout le tableau, donc de la position `0` à la position `n - 1`. On calcule la position moyenne `m` entre ces deux positions et on vérifie si la valeur qu'on cherche est dans la position `m`. Si c'est le cas, on termine en retournant `m` comme résultat.

Si ce n'est pas le cas, on continue de chercher, soit dans la moitié du tableau qui reste à la gauche de la position `m` (si la valeur qu'on cherche est inférieure à l'élément du tableau en position `m`), soit dans la moitié droite (si elle est supérieure).

On continue de chercher jusqu'à trouver la valeur au milieu d'un sous-tableau ; si on arrive à éliminer tous les éléments du tableau sans l'avoir trouvée, on retournera `-1` pour indiquer que la valeur n'apparaît pas dans le tableau.

Comme on a vu en cours d'Introduction à l'informatique, cet algorithme élimine à chaque étape à peu près la moitié du tableau, ce qui signifie qu'il termine en log n étapes ou, en symboles, en temps O(log n).

Question 3 : Écrire le code de la fonction `binary_search`. Comme pour `linear_search`, cette fonction doit retourner un entier (entre `0` et `len(a) - 1`) qui correspond à la position de l'élément `x` dans le tableau `a`, ou bien l'entier `-1` si `x` n'apparaît pas dans `a`. Utiliser l'algorithme de recherche dichotomique décrit ci-dessus.

Question 4 : Pour tester la fonction `linear_search`, afficher les résultats de la recherche des valeurs `2`, `5`, `10` et `6` dans le tableau trié `a2`.

Comparaison des temps de calcul

Pour mesurer le temps de calcul d'un morceau de code en Python on peut utiliser la fonction `default_timer()`, qui retourne un `float` representant le temps actuel en secondes :

```
start = default_timer()
# insérer le code ici
end = default_timer()
```

On obtiendra (une approximation du) temps d'execution du code en calculant `end - start`, le difference entre le temps initial et final.

Pour comparer le temps de calcul des fonctions `linear_search` et `binary_search` et apprecier la différence entre un algorithme à temps linéaire et un algorithme à temps logarithmique, on va mesurer combien de seconds prennent les deux fonctions dans le pire des cas (c'est-à-dire, quand la valeur cherchée n'apparait pas dans le tableau) pour un ensemble de tableaux de differentes longeurs.

Question 5 : Dans le tableau `lengths` on a accumulé les entiers de `1` à `1000` ; pour chaque longueur `n` on crée un tableau `a` contenant les entiers de `0` à `n - 1` avec la fonction `list(range(n))`, qui convertit le `range` en tableau.

Mesurer le temps du calcul des deux fonctions `linear_search` et `binary_search` sur chaque tableau `a` dans le pire des cas (c'est-à-dire, en cherchant une valeure qui n'apparait pas) et ajouter les temps obtenus respectivement aux tableaux `times_linear` et `times_binary`.

Le code après la boucle `for` dessinnera un graphe `matplotlib` avec les temps de calcul mesurés.

Correction

```
from matplotlib.pyplot import plot, show, legend
from timeit import default_timer


#Q1

def linear_search(x, a):
    for i in range(len(a)):
        if a[i] == x:
            return i
    return -1


#Q2

print("Recherche linéaire:")

a1 = [5, 1, 3, 2, 7, 9, 4, 8, 0, 6]

print(linear_search(2, a1))
print(linear_search(5, a1))
print(linear_search(10, a1))
print(linear_search(6, a1))


#Q3

def binary_search(x, a):
    left = 0
    right = len(a) - 1
    while left <= right:
        m = (left + right) // 2
        if x == a[m]:
            # élément trouvé
            return m
        elif x < a[m]:
            # on cherche dans la moitié gauche
            right = m - 1
        else:
            # on cherche dans la moitié droite
            left = m + 1
    # élément absent
    return -1
```

```
#Q4

print("Recherche dichotomique:")

a2 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(binary_search(2, a2))
print(binary_search(5, a2))
print(binary_search(10, a2))
print(binary_search(6, a2))


#Q5

# lengths = liste des longueurs des tableaux
lengths = []
for i in range(1, 1001):
    lengths.append(i)

times_linear = []
times_binary = []

for n in lengths:
    a = list(range(n))
    start = default_timer()
    result = linear_search(-1, a)
    end = default_timer()
    times_linear.append(end - start)
    start = default_timer()
    result = binary_search(-1, a)
    end = default_timer()
    times_binary.append(end - start)

if __name__ == '__main__':
    plot(lengths, times_linear, label="linear")
    plot(lengths, times_binary, label="binary")
    legend()
    show()
```

Partie 2 — Tri par sélection

Ex. 2

Il existe de nombreux algorithmes pour trier les tableaux (par exemple, le tri par insertion et le tri fusion qu'on a vu dans l'UE Introduction à l'informatique).

Le tri par sélection d'un tableau `a` consiste en chercher le minimum de `a`, en échanger cet élément avec l'élément en position `0` et en répéter cette procédure pour le sous-tableau de la position `1` à la position `n - 1`, puis pour le le sous-tableau de la position `2` à la position `n - 1`, etc., jusqu'à la fin du tableau.

Après l'`i`-ème étape de l'algorithme, la portion initiale du tableau (de la position `0` à la position `i`) sera trié, ce qui nous garantit que la totalité du tableau sera triè à la fin de l'algorithme.

Question 1 : Écrire la fonction `find_minimum(a, left, right)`, qui prend comme arguments un tableau `a` (de longueur `>= 1`) et deux entiers `left` et `right` et retourne la position de l'élément minimum du sous-tableau de la position `left` à la position `right`.

Question 2 : Affecter à la variable `m` le minimum du tableau `a1` avec la fonction `find_minimum`.

Question 3 : Écrire la fonction `swap(a, i, j)`, qui prend commme arguments un tableau `a` et deux entiers `i` et `j` et qui échange les éléments de position `i` et `j` en `a`. (La fonction ne retourne aucun résultat, puisque sa tâche est de modifier directement le tableau `a`.)

Question 4 : Échanger les éléments en position `1` et `7`, puis ceux en position `4` et `3` dans le tableau `a1` à l'aide de la fonction `swap`. Afficher le tableau `a1` ainsi modifié.

Question 5 : Écrire la fonction `selection_sort(a)`, qui prend comme argument un tableau `a` et le trie avec l'algorithme de tri par sélection. Utiliser les fonctions `find_minimum` et `swap` pour résoudre cet exercice. (La fonction ne retourne aucun résultat, puisque sa tâche est de modifier directement le tableau `a`.)

Question 6 : Trier le tableau `a1` à l'aide de la fonction `selection_sort` qu'on vient d'écrire. Afficher le tableau `a1` ainsi modifié.

Question 7 : Évaluer expérimentalement l'efficacité de l'algorithme de tri par sélection. Procéder comme dans le dernier exercice sur la recherche dans lex tableaux, en générant un tableau `a` de taille `n` pour chaque valeur de `n` dans un intervalle approprié (conseil : commencer avec un petit intervalle, par exemple de 1 à 100, puisque l'algorithme de tri est beaucoup plus lent que les algorithmes de recherche).

Utiliser la fonction `shuffle(a)` pour permuter de façon aléatoire le contenu

du tableau `a`. Mesurer le temps de calcul de `selection_sort(a)` pour chaque tableau `a` et dessiner un graphe qui montre le temps pour chaque taille de tableau.

Après avoir analysé le graphe et le code, quelles conclusions peut-on tirer sur le temps de calcul de `selection_sort` en fonction de la taille des entrées ?

Correction

```
from matplotlib.pyplot import plot, legend, show
from timeit import default_timer
from random import shuffle


#Q1

def find_minimum(a, left, right):
    i = left
    for j in range(left + 1, right + 1):
        if a[j] < a[i]:
            i = j
    return i


#Q2

a1 = [9, 4, 1, 3, 0, 8, 2, 6, 7, 5]
print(a1)
m = find_minimum(a1, 0, len(a1)-1)
print("Le minimum est en position", m)
print()


#Q3

def swap(a, i, j):
    tmp = a[i]
    a[i] = a[j]
    a[j] = tmp


#Q4

print("Échange d'éléments")
```

```
swap(a1, 1, 7)
swap(a1, 4, 3)
print(a1)
print()

#Q5

def selection_sort(a):
    n = len(a)
    for i in range(n-1):
        m = find_minimum(a, i, n - 1)
        swap(a, i, m)


#Q6

print("Tri par selection")
selection_sort(a1)
print(a1)

#Q7

lengths = []
for i in range(1, 100):
    lengths.append(i)
times = []

for n in lengths:
    a = list(range(n))
    shuffle(a)
    start = default_timer()
    selection_sort(a)
    end = default_timer()
    times.append(end - start)

if __name__ == '__main__':
    plot(lengths, times)
    show()
```