Rechercher

INFORMATIQUE COMMUNE - TP nº 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 :

```python
from random import randint

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

Fonction RECHERCHE_SÉQUENTIELLE(t, elem)
    n ← taille(t)
    pour i de 0 à $n-1$ **répéter**
        si t[i]= elem **alors**
            renvoyer i ▷ élément trouvé, on renvoie sa position dans t
    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. Il faudra $C(n) = kn$ opérations élémentaires pour réaliser la recherche séquentielle : le coût est proportionnel à la taille du tableau.

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 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**

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

B2. 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 itérations, le tableau considéré par les indices `g` et `d` ne contient plus qu'un ou deux éléments. Supposons qu'il n'en reste qu'un seul [1].

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.

B3. Tracer le graphique des temps d'exécution des algorithmes précédents en fonction de n, la taille du tableau. Les tracés sont-ils cohérents avec les calculs des coûts effectués précédemment ? On prendre des tailles de tableau n de 10 à 100000 exclu par pas de 500.

Solution :

```python
from matplotlib import pyplot as plt

def search_timing():
    sizes = [i for i in range(10, 100000, 500)]
    M = 5 * max(sizes)
    results = []
    for i in range(len(sizes)):
        t = sorted([randrange(0, M+1) for _ in range(sizes[i])])
        mem_t = t[:]
        results.append([])
        for method in [seq_search, dichotomic_search]:
            t = mem_t[:]
            tic = time.perf_counter()
            method(t, M // 4)
            toc = time.perf_counter()
            results[i].append(toc - tic)

    seq = [results[i][0] for i in range(len(sizes))]
    dics = [results[i][1] for i in range(len(sizes))]

    plt.figure()
    plt.plot(sizes, seq, color='cyan', label='Sequential search')
    plt.plot(sizes, dics, color='blue', label='Dichotomic search')
    plt.xlabel('n', fontsize=18)
    plt.ylabel('time', fontsize=16)
    plt.legend()
    plt.show()
```

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 programmé 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. Construire une liste imbriquée de listes d'entiers choisis aléatoirement en 0 et 100 exclu sont la taille des sous-listes est variable.

Solution :

```
from random import randrange

h = 5
L=[]
for i in range(h):
    L.append([])
    for j in range(randrange(h)):
        L[i].append(randrange(0, 100))
```

C2. 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
```

C3. Appliquer la recherche dichomotique une liste imbriquée en la mettant à plat et en la triant.

Solution :

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

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

Solution :

```
def n_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 `size_sl(L : list[list[int]])-> list[int]` qui renvoie la liste des taille 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 size_sl(L: list[list[int]]) -> list[int]:
    sizes = []
    for i in range(len(L)):
        sizes.append(len(L[i]))
    return sizes
```