TP INFORMATIQUE

Recherches séquentielles, recherches dichotomiques

Première approche

Quelques fonctions de recherche

À l'aide de l'instruction suivante, générer une liste `L` contenant 100 éléments. Chaque élément est aléatoirement un entier compris entre 0 et 20. Il peut donc y avoir plusieurs éléments (à des indices différents) de même valeur.

```
import random as rd
L = [ rd.randint(0,21) for i in range(100)]
```

1. Écrire une fonction `RechercheS(x,L)` d'argument un entier `x` et la liste `L` qui retourne `True` si `x` est présent dans `L`, `False` sinon.
2. Écrire une fonction `RecherchePos(x,L)` de mêmes arguments qui retourne l'indice d'apparition de `x` s'il est présent, `False` sinon.
3. Écrire une fonction `RechercheMax(L)` d'argument la liste `L` qui retourne *sans utiliser la fonction native max de python* l'élément maximal de la liste.

Une première amélioration

Revenons sur la recherche d'un élément dans la liste : l'algorithme écrit précédemment nécessite de parcourir tous les éléments de la liste un par un et de les comparer à l'élément recherché. On parle de recherche séquentielle dont la durée d'exécution croit linéairement en fonction du nombre d'éléments dans la liste.

On peut améliorer les choses lorsque la liste initiale est triée. On va l'illustrer sur la liste suivante :

```
import random as rd
L = [rd.randint(1,5)]
for i in range(99):
    L.append(L[-1] + rd.randint(1,5))
```

1. Écrire une fonction `RechercheC(x,L)` d'argument un entier `x` et la liste `L` qui retourne `True` si `x` est présent dans `L`, `False` sinon, mais qui ne nécessite pas de parcourir toute la liste, sauf dans le cas où `x` est supérieur au dernier élément de la liste.

Recherche par une méthode dichotomique

L'intérêt de la méthode dichotomique est d'accélérer grandement la recherche de l'élément. Cette méthode ne fonctionne **QUE** si la liste initiale est triée (on verra des exemples d'algorithme de tri au cours de l'année).

Le principe est de comparer l'élément à celui se trouvant « au milieu » du tableau. S'ils sont égaux c'est terminé. Sinon on recommence avec la partie du tableau pouvant contenir l'élément. Sur l'exemple suivant on recherche l'élément 28 :

				↓						
15	16	18	19	23	24	28	29	31	33	$28 > 23$
							↓			
15	16	18	19	23	24	28	29	31	33	$28 < 29$
					↓					
15	16	18	19	23	24	28	29	31	33	$28 > 24$
						↓				
15	16	18	19	23	24	28	29	31	33	$28 = 28$

On constate que l'élément est trouvé en 4 étapes. Alors qu'on en a besoin de 7 en utlisant la méthode de l'exercice précédent. Si l'élément n'est pas présent l'algorithme fonctionne également comme illustré ci-dessous (pour la recherche de 27) :

				↓						
15	16	18	19	23	24	28	29	31	33	$27 > 23$
							↓			
15	16	18	19	23	24	28	29	31	33	$27 < 29$
					↓					
15	16	18	19	23	24	28	29	31	33	$27 > 24$
						↓				
15	16	18	19	23	24	28	29	31	33	$27 \neq 28$

Là encore on peut conclure plus rapidement par cette méthode.

On peut montrer que pour une liste de N éléments, il faut en ordre de grandeur $\log N$ étapes par la méthode dichotomique contre N par la méthode séquentielle (que l'élément soit présent ou non).

Version itérative

L'algorithme consiste à continuer à rechercher tant que c'est possible, à savoir tant qu'il reste des éléments à chercher. Pour ce faire à chaque fois qu'on coupe le tableau en deux, on identifie ses indices extrêmes et on continue tant qu'ils ne sont pas égaux. Lorsqu'ils deviennent égaux il ne reste plus qu'un élément à tester.

1. En se basant sur l'explication précédente, compléter le code suivant en remplaçant les ? ? par ce qu'il faut.

```
def dichotomie(x,L):
    """ Recherche la présence de l' élément x dans la liste L
        préalablement triée. Si x est present retourne True,
        sinon retourne False. """

      g , d = 0 , len(L) -1 # initialisation

      while g <=d :
            m = (g +d) // 2 # élément a tester

            if L[m]  == x :
                return ??
            elif L[m] < x  :
                ??
            else :
                 ??
      return ??
```

2. Il est en pratique inutile d'exécuter la fonction si $x < \mathrm{L}[0]$ ou $x > \mathrm{L}[-1]$. Ajouter un `assert` à la fonction précédente en conséquence.

3. Analyser le fonctionnement de votre fonction sur la liste présentée en introduction de la méthode (pour la recherche de 28 et de 27).
4. Tester votre fonction sur les cas suivants : un élément `x` présent avec une liste contenant un nombre pair de termes, puis un nombre impair, un élément `x` absent de la liste.
5. On propose la deuxième version suivante :

```
def dichotomie(x,L):
    g , d = 0 , len(L)
    while g  < d - 1 :
        k = (g + d)//2
        if x < L[k] :
            d=  k
        else  :
            g = k
    return x == liste[g]
```

6. Analyser le fonctionnement de cette fonction sur la liste présentée en introduction de la méthode (pour la recherche de 28 et de 27). Comparer à la première version.

> Pour trier une liste on peut utiliser la fonction `sorted` native de python, qui prend pour argument une liste à trier et retourne la liste triée. On peut aussi utiliser la méthode `sort` qui modifie une liste `L` existante en écrivant `L.sorted()`

Version récursive

La méthode par dichotomie se prête bien à une écriture récursive puisqu'à chaque étape on repose la même question (avec un nombre d'éléments divisé par deux).

Compléter la fonction suivante, puis la tester comme précédemment.

```
def dichotomieR(x,L):
    """ Recherche de l'élément x dans L par méthode récursive """

    m = len(L) //2

    if m == 0 :  # Condition d'arrêt
        return ??

    else:
        if x ==L[m] :
            ??
        elif x > L[m]:
            ??
        else :
            ??
```

Des exercices pour les les plus rapides

1. Écrire une fonction `RechercheMax2(L)` d'argument la liste `L` qui retourne les deux premiers maxima de la liste.
2. Écrire une fonction `insertion(x,L)` d'argument un nombre `x` et une liste `L` supposée triée qui ne fait rien si `x` est contenu dans `L` et modifie la liste `L` en plaçant `x` à la bonne place sinon.
3. Écrire une fonction `TempsCalcul(x,L,methode)` qui retourne la durée d'une recherche. L'argument `methode` est soit `"sequentiel"` soit `"dichotomie"`. Vérifier alors la conjecture d'une durée d'exécution en N pour la recherche séquentielle et en $\log N$ pour la recherche dichotomique. On pourra utiliser la fonction `perf_counter` du module `time`.