Algorithmes et Pensée Computationnelle

Algorithmes de recherche - Exercices de base

Le but de cette séance est de se familiariser avec les algorithmes de recherche. Dans cette série d'exercices, nous manipulerons des listes et collections en Java et Python. Nous reviendrons sur la notion de récursivité et découvrirons les arbres de recherche. Au terme de cette séance, l'étudiant sera en mesure d'effectuer des recherches de façon efficiente sur un ensemble de données.
Le code présenté dans les énoncés se trouve sur Moodle, dans le dossier **Code**.

1 Recherche séquentielle (ou recherche linéaire)

1.1 Définition

Une recherche **séquentielle** (ou **linéaire**) est une méthode permettant de trouver un élément dans un ensemble de données (liste, tableau ou dictionnaire). Elle vérifie un à un chaque élément de gauche à droite jusqu'à ce qu'une correspondance soit trouvée ou que toute la liste ait été parcourue.

Si l'élément recherché est trouvé, l'algorithme **renvoie l'index**, c'est-à-dire la position, de l'élément dans l'ensemble de données.

Sa complexité dans le pire des cas est de **O(n)** correspondant à la longueur de l'ensemble de données, et dans le meilleur des cas de **O(1)**, lorsque l'élément se trouve en première position. Dans la pratique, l'algorithme de recherche séquentielle n'est pas couramment utilisé eu égard de sa complexité élevée et des alternatives de recherche plus efficaces comme la recherche binaire.

1.2 Exercices

Question 1: (*10 minutes*) Recherche séquentielle - 1 (Python)

À partir des éléments ci-dessous, écrivez une fonction qui cherche **x** dans la liste **L**.
La fonction doit retourner l'index de l'élément correspondant de la liste si **x** est dans la liste et "-1" si **x** n'est pas dans la liste (avec **x = 100**).

Python :

```
1  def recherche_sequentielle(liste, x):
2      #complètez ici
3
4  L = [3,55,6,8,3,5,56,33,6,5,3,2,99,53,532,75,21,963,100,445,56,56,24]
5  x = 100
6
7  resultat = recherche_sequentielle(L, x)
8  print(resultat)
```

Conseil

Définissez votre fonction de recherche linéaire en utilisant une boucle **for** ou une boucle **while**.

Attention : la fonction doit retourner l'index de la valeur et non pas la valeur. Pour cela, pensez à utiliser **range(len(list))** avec la boucle **for** et une incrémentation **"i = i+1"** avec la boucle **while**.

Utilisez la fonction **print()** pour afficher l'index lorsque vous l'aurez trouvé et un autre message le cas échéant.

Solution

Python :

```python
#Définition de la fonction
def recherche_sequentielle(liste, x):
    for index in range(len(liste)):  # i représente l'index
        if liste[index] == x:
            print("X est présent dans la liste à l'index :", index)
            return index

    print("X n'est pas présent dans la liste")
    return -1

# Déclaration de la liste et de la variable x
L = [3, 55, 6, 8, 3, 5, 56, 33, 6, 5, 3, 2, 99, 53, 532, 75, 21, 963, 100, 445, 56, 45, 12, 56, 24]
e = 100

# Exécution de l'algorithme
resultat = recherche_sequentielle(L,e)
print(resultat)
```

Question 2: (*10 minutes*) Recherche séquentielle - 2 (Python)

Soit une liste d'entiers **non triée** `L` ainsi qu'un entier `e`. Écrivez un programme qui retourne l'élément de la liste `L` dont la valeur est la plus proche de `e` en utilisant une recherche séquentielle.

Exemple :
L = [16, 2, 25, 8, 12, 31, 2, 56, 58, 63]
e = 50
Résultat attendu : 56

Au cas où deux éléments se trouveraient à équidistance de `e`, renvoyez l'élément le plus petit.

Python :

```python
#Definition de la fonction ayant pour argument une liste et un nombre
def plus_proche_sequentielle(list,nb):
    diff = -1 #Initialisation de la variable (-1 car les différence calculée après seront toujours positives)
    resultat = None #Initialisation de la variable pour le résultat

    #Complètez ici

#Déclaration de la liste et de la variable e
L = [16, 2, 25, 8, 12, 31, 2, 56, 58, 63]
e = 50

#Exécution de la fonction
resultat = plus_proche_sequentielle(L,e)
print(resultat)
```

Conseil

Complétez la fonction **plus_proche_sequentielle** et exécutez le code.

Utilisez les valeurs absolues pour comparer les différences, la plus petite pouvant être positive ou négative. En Python, la fonction `abs()` retourne la valeur absolue. Exemple : **`abs(3-10)` retourne 7.**

Étant donné que la liste est **non triée**, l'algorithme doit obligatoirement la parcourir intégralement.

L'algorithme doit calculer la différence entre `e` et chaque élément de la liste `L` en gardant toujours la plus petite différence trouvée. À la fin, il retourne l'élément de la liste correspondant à la plus petite différence.

Solution

Python :

```
#Definition de la fonction ayant pour argument une liste et un nombre
def plus_proche_sequentielle(liste,nb):
    diff = -1 #Initialisation de la variable ( -1 car les différences calculées après seront toujours positives)
    resultat = None #Initialisation de la variable pour le résultat

    #Solution
    for elem in liste:
        if diff == -1 or abs(elem-nb) < diff:
            diff = abs(elem-nb) #new diff #
            resultat = elem
        elif (abs(elem-nb) == diff):
            resultat=min(elem, resultat)

    return resultat

#Déclaration de la liste L et de la variable e
L = [16, 2, 25, 8, 12, 31, 2, 56, 58, 63]
e = 50

#Exécution de la fonction
resultat = plus_proche_sequentielle(L,e)
print(resultat)
```

Question 3: (*10 minutes*) Recherche séquentielle - 3 (Python)
Considérez une **liste d'entiers triés** `L` ainsi qu'un entier `e`. Écrivez un programme qui retourne l'index de l'élément `e` de la liste `L` en utilisant une recherche séquentielle. Si `e` n'est pas dans `L`, retournez -1.

Exemple

L = [123,321,328,472,549]
e = 328
Résultat attendu : 2

```
def recherche_sequentielle(L,e):
    for elem in L: #Ici, i correspond à la valeur et non l'index.
        #complètez ici

    L = [123,321,328,472,549]
    e = 328
    resultat = recherche_sequentielle(L,e)
    print(resultat)
```

Conseil

Une liste triée permet une recherche plus efficace à l'aide d'un algorithme plus simple.
Retournez l'index de la valeur dans la liste. Pensez à utiliser la fonction **index()** qui retourne l'index d'un élément au sein d'une liste en Python.
Exemple et syntaxe : **lst.index(i)** va indiqué la position de l'élément **i** dans la liste **lst**.

Solution

Python :

```
1  def recherche_sequentielle(L,e):
2      for elem in L:
3          #Solution
4          if elem == e:
5              return L.index(elem) #L'algorithme prend fin aussitot que la valeur recherchée est trouvée.
6      return -1 # Si la valeur n'a pas été trouvée, la fonction retourne -1
7
8  L = [123,321,328,472,549]
9  e = 328
10 resultat = recherche_sequentielle(L,e)
11 print(resultat)
```

2 Recherche binaire

2.1 Définition

Le but de la recherche binaire est de trouver l'élément recherché de façon optimale. Pour cela, il est nécessaire d'utiliser **une liste d'éléments triés**.

La complexité de l'algorithme de recherche binaire est **O(log n)**. Cependant, il ne faut pas oublier le coût lié à l'obtention d'une liste triée à partir d'une liste non triée.

L'algorithme de recherche binaire divise l'intervalle de recherche par deux à chaque itération jusqu'à ce qu'il trouve l'élément **x** ou que l'intervalle soit vide.

Ainsi, si **x** est plus petit que l'élément du milieu, l'algorithme va choisir la moitié de gauche comme intervalle de recherche et ainsi de suite. Si **x** est plus grand que l'élément du milieu la recherche va se faire dans la moitié droite de l'intervalle.

La recherche binaire se base sur les comparaisons d'ordre, alors que la recherche séquentielle se base les comparaisons d'égalité pour trouver l'élément recherché.

2.2 La récursivité

Une fonction récursive est une fonction qui s'appelle elle-même pendant son exécution. Vous trouverez ci-dessous un exemple de fonction récursive utilisée pour effectuer un calcul factoriel.

(Rappel : 4 ! = 4 x 3 x 2 x 1 = 24)

```
1  def factoriel(n):
2      if n == 1:
3          return n
4      else:
5          return n * factoriel(n - 1)
6
7  #Exécution de la fonction
8  factorielle(4)
9  #La fonction retourne 24
```

Les fonctions récursives sont courantes en informatique car elles permettent aux programmeurs d'écrire des programmes efficaces en utilisant une quantité minimale de code. Leur principal inconvénient est le fait qu'elles peuvent provoquer des exécutions infinies et d'autres résultats inattendus si elles ne sont pas écrites correctement. Si la fonction n'inclut pas les cas permettant d'arrêter la récursivité, celle-ci se répétera à l'infini, provoquant le plantage du programme ou, pire encore, l'arrêt de tout le système informatique.

2.3 Exercices

Question 4: (🕒 *10 minutes*) Propriété des arbres binaires de recherche

Supposons que nous ayons des nombres entre 1 et 1000 dans un arbre binaire de recherche, et que nous voulions rechercher le nombre 363. Lesquelles des séquences suivantes ne pourraient pas être une séquence des nœuds examinés avant de trouver 363 ?

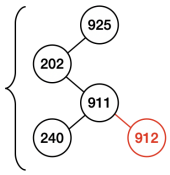
1. 2, 252, 401, 398, 330, 344, 397, 363.
2. 924, 220, 911, 244, 898, 258, 362, 363.
3. 925, 202, 911, 240, 912, 245, 363.
4. 2, 399, 387, 219, 266, 382, 381, 278, 363.
5. 935, 278, 347, 621, 299, 392, 358, 363.

Solution

✔ 2, 252, 401, 398, 330, 344, 397, 363.

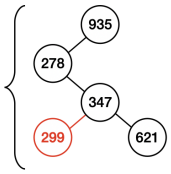
✔ 924, 220, 911, 244, 898, 258, 362, 363.

✘ 925, 202, 911, 240, 912, 245, 363. →



✔ 2, 399, 387, 219, 266, 382, 381, 278, 363.

✘ 935, 278, 347, 621, 299, 392, 358, 363. →



Question 5: (*20 minutes*) Récursivité et itération (Python)

À partir de la liste d'éléments **triés** ci-dessous, écrivez premièrement **une fonction récursive** puis **une fonction itérative** qui cherche `x` dans la liste. La fonction doit retourner **l'index** de l'élément correspondant de la liste si `x` est dans la liste et `-1` dans le cas contraire.
Ici, `x = 5`.

Question 5.1 - Version récursive

```
#Version récursive
def recherche_binaire_recursive(liste,s,r,x):
    #complètez ici

L=[1,3,4,5,7,8,9,15]
s = 0
r = len(L)
x = 5
recherche_binaire_recursive(L,s, r, x)
```

Question 5.2 - Version itérative

```
#Version itérative
    def recherche_binaire_iterative(liste,s,r,x):
    #complètez ici

L = [1,3,4,5,7,8,9,15]
s = 0
r = len(L)-1
x = 7
recherche_binaire_iterative(L,s, r, x)
```

Conseil

Complétez la fonction récursive **recherche_binaire_recursive** et la fonction itérative **recherche_binaire_iterative**.

Détails sur les arguments de la fonction :
L : La liste dans laquelle nous effectuons la recherche.
s : L' élément de départ de la partie de la liste à fouiller (0 au départ).
r : Le dernier élément de la partie de la liste à fouiller (len(liste) au départ).
x : La valeur recherchée.

Ainsi, à chaque itération, vos fonctions vont modifier les valeurs de base données en argument pour resserrer l'intervalle jusqu'à trouver la valeur recherchée.
Pour définir le milieu d'un intervalle qui contient un nombre pair ou impair d'éléments, divisez l'ensemble en 2 et utilisez la fonction **int()** pour convertir le résultat en entier.
Exemple :
liste1 = [1,2,3,4,5]
s = 0
r = len(liste1)
Calcul du milieu de l'intervalle :
(s+r)/2) = (0+5)/2 = 2.5 #Pas de correspondance
int((s+r)/2) = int((0+5)/2) = int(2.5) = 2
Arrondi vers le bas

Pour la version itérative, il est conseillé d'utiliser une boucle **while**.

Solution

5.1 - Version récursive

```
1
2  def recherche_binaire_recursive(L, s, r, x):
3      if r >= s:
4          mid = int((s + r)/2)
5          print(f'Le milieu de la liste {L} est {L[mid]} situé à l\'indice {mid}')
6          if L[mid] == x:
7              return mid
8          elif L[mid] > x:
9              return recherche_binaire_recursive(L, s, mid-1, x)
10         else:
11             # on peut aussi utiliser mid mais mid+1 evite une comparaison
12             # de plus car cette comparaison est faite en amont
13             return recherche_binaire_recursive(L, mid+1, r, x)
14     else:
15         return -1
16
17 L=[1,3,4,5,7,8,9,15]
18 s = 0
19 r = len(L)-1 #8 --> première moitié = 4 -->"7" in L --> 7>5 --> deuxième moitié = (0+4)/2 = 2, etc.
20 x = 8
21 print(recherche_binaire_recursive(L,s, r, x))
```

Solution

5.2 - Version itérative

```
1  def recherche_binaire_iterative(liste,s,r,x):
2      while s <= r:
3          mid = int(s + (r-s)/2)
4          print(f"La moitié correspond à {mid}")
5          # Si l'élément du milieu correspond à x, on retourne mid
6          if liste[mid] == x:
7              print("X dans liste à l'index: ", mid)
8              return mid
9          # Si x est plus grand, on ignore la moitié de gauche
10         elif liste[mid] < x:
11             s = mid+1
12         # Si x est plus petit, on ignore la moitié de droite
13         else:
14             r= mid-1
15     # Si on sort de la boucle, c'est que l'élément est absent de la liste
16     print("X absent de liste")
17     return -1
18
19 L = [1,3,4,5,7,8,9,15]
20 s = 0
21 r = len(L)
22 print(f"La liste contient {r} éléments")
23 x = 6
24 recherche_binaire_iterative(L,s, r, x)
```

3 Arbre de recherche binaire

3.1 Définition

Un arbre de recherche binaire est une structure de données au même titre que les listes, tuples, dictionnaires, etc. Leur particularité est qu'au lieu d'ordonner les éléments les uns à la suite des autres, les arbres de recherche binaire enregistrent les valeurs de manière relationnelle.

En effet, l'arbre est divisé en branches qui peuvent elles-même contenir deux branches (enfants) et ainsi de suite. Lorsqu'une branche n'a pas de branches subséquentes (enfants), on parle de feuille.

Les branches peuvent donc contenir de 0 à 2 autres branches. On parle alors de branche de gauche et de celle de droite. La branche de gauche est forcément plus petite que la branche parente et celle de droite est forcément plus grande.

De cette façon, on garantit que l'arbre est toujours ordonné même en rajoutant ou en retirant des éléments

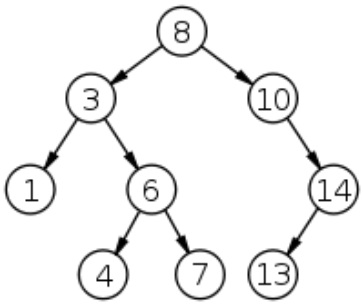


FIGURE 1 – Exemple d'arbre binaire

3.2 Exercices

Question 6: (🕒 *20 minutes*) Recherche dans un arbre - Python

Dans l'exercice suivant, nous vous donnons un arbre binaire avec les mêmes valeurs que le schéma précédent et dont la racine est la variable **root**.

Écrivez une fonction qui retourne **True** si la valeur **value** se trouve dans l'arbre et **False** dans le cas contraire. Vous pouvez utiliser la variable **value** pour afficher la valeur d'un nœud (**node**) et les variables **left** et **right** sur les **nodes** pour accéder aux branches enfants, utilisez **node.value**, **node.left** et **node.right**.

Complétez la fonction **recherche_arbre**.

```
1  import sys
2  import traceback
3
4  def recherche_arbre(node, value):
5      # Complétez ici
6
7
8  #Attention à inclure les lignes de codes additionnelles présentes dans le fichier "question6.py" sur Moodle
9
```

Conseil

Indice : Utilisez une fonction récursive.

Pour cette question, vous devez télécharger le fichier **“question6.py”** sur Moodle et copier tout son contenu dans votre fichier Python. Celui-ci contient les classes permettant de définir les arbres selon le concept vu en cours.

Il contient également un code de “vérification”. Celui-ci va automatiquement tester votre fonction avec différent paramètres et vérifier si les résultats obtenus correspondent au résultats attendus.

Solution

```
1  import sys
2  import traceback
3
4
5  def recherche_arbre(node, value):
6      #Solution
7      if node == None:
8          return False
9
10     if node.value == value:
11         return True
12     if node.value > value:
13         return recherche_arbre(node.left,value)
14     else:
15         return recherche_arbre(node.right,value)
```