C8 Diviser pour régner

1. Activités

1.1.

■

Activité 1 : *Retour sur l'algorithme de dichotomie*

Aide

Cette activité revient sur deux algorithmes de recherche d'un élément dans une liste déjà rencontrés en classe de première.

1. Ecrire une fonction `recherche(x,l)` qui en effectuant un parcours simple de la liste, renvoie `True` ou `False` selon que l'élément `x` se trouve ou non dans la liste `l`.
2. On suppose maintenant que la **liste est triée**, l'algorithme de recherche par dichotomie vue en classe de première consiste alors à

   1
   partager la liste en deux listes de longueurs égales (à une unité près)

   2
   comparer l'élément recherché avec celui situé au milieu de la liste

   3
   en déduire dans quelle moitié poursuivre la recherche
   On s'arrête lorsque la zone de recherche ne contient plus qu'un élément.

   a. Faire fonctionner "à la main" cet algorithme pour rechercher `6` dans `[1,3,5,7,11,13]`.

   b. Programmer cet algorithme en version impérative.

A compléter - Dichotomie version impérative

❤

```
def recherche_dichotomique(tab, val) :
    '''
    renvoie True ou False suivant la présence de la valeur val dans le tableau trié tab.
    '''
    i_debut = 0
    i_fin = len(tab) - 1
    while i_debut <= i_fin :
        i_centre = (... + ...) // 2     # (1)
        val_centrale = tab[...]         # (2)
        if val_centrale == val:             # (3)
            return True
        if val_centrale < val:              # (4)
            i_debut = ....              # (5)
        else :
            i_fin = ...
    return False
```

a. on prend l'indice central

b. on prend la valeur centrale

c. si la valeur centrale est la valeur cherchée...

d. si la valeur centrale est trop petite...

e. on ne prend pas la valeur centrale qui a déjà été testée

Exemple d'utilisation :

Script Python

```
>>> tab = [1, 5, 7, 9, 12, 13]
>>> recherche_dichotomique(tab, 12)
True
>>> recherche_dichotomique(tab, 17)
False
```

À chaque tour de la boucle `while`, la taille de la liste est divisée par 2. Ceci confère à cet algorithme une **complexité logarithmique** (bien meilleure qu'une complexité linéaire).

Complexité

Pour pouvoir majorer le nombre maximum d'itérations, si le tableau contient n valeurs, et si on a un entier k tel que $n \leq 2^k$, alors puisque qu'à chaque itération, on sélectionne une moitié de ce qui reste :

- au bout d'une itération, une moitié de tableau aura au plus $\dfrac{2^k}{2} = 2^{k-1}$ éléments,
- un quart aura au plus 2^{k-2}
- et au bout de i itérations, la taille de ce qui reste à étudier est de taille au plus 2^{k-i}.
- En particulier, si l'on fait k itérations, il reste au plus $2^{k-k} = 1$ valeur du tableau à examiner. On est sûr de s'arrêter cette fois-ci

On a donc montré que si l'entier k vérifie $n \leq 2^k$, alors l'algorithme va effectuer au plus n itérations. La plus petite valeur est obtenue pour $\log(n) = \log_2 k$.
Ainsi, la complexité de la fonction est de l'ordre du logarithme de la longueur de la liste ($O(log_2(n))$).

Donc l'algorithme du tri fusion a une complexité de l'ordre de $n \times log_2(n)$.

3. Dichotomie récursive sans slicing

Il est possible de programmer de manière récursive la recherche dichotomique sans toucher à la liste, et donc en jouant uniquement sur les indices :

Dichotomie version récursive sans slicing

❤

```
1   def dicho_rec_2(tab, val, i=0, j=None): # (1)
2       if j is None:                       # (2)
3           j = len(tab)-1
4       if i > j :
5           return False
6       m = (i + j) // 2
7       if tab[m] < val :
8           return dicho_rec_2(tab, val, m + 1, j)
9       elif tab[m] > val :
10          return dicho_rec_2(tab, val, i, m - 1 )
11      else :
12          return True
```

1. Pour pouvoir appeler simplement la fonction sans avoir à préciser les indices, on leur donne des paramètres par défaut.
2. Il est impossible de donner `j=len(tab)-1` par défaut (car `tab` est aussi un paramètre). On passe donc par une autre valeur (ici `None`) qu'on va ici intercepter.

Exemple d'utilisation :

🐍 **Script Python**

```
>>> tab = [1, 5, 7, 9, 12, 13]
>>> dicho_rec_2(tab, 12)
True
>>> dicho_rec_2(tab, 17)
False
```

Les algorithmes de dichotomie présentés ci-dessous ont tous en commun de diviser par deux la taille des données de travail à chaque étape. Cette méthode de résolution d'un problème est connue sous le nom de *diviser pour régner,* ou *divide and conquer* en anglais.

Une définition pourrait être :

Définition

❤

Un problème peut se résoudre en employant le paradigme *diviser pour régner* lorsque :
- il est possible de décomposer ce problème en sous-problèmes **indépendants**.
- la taille de ces sous-problèmes est une **fraction** du problème initial

Remarques :

- Les sous-problèmes peuvent nécessiter d'être ensuite recombinés entre eux (voir plus loin le tri fusion).

1.2.

■

Activité 2 : *Tri fusion*

1. Algorithmes de tri vus en première et revus cette année :
   a. Rappeler rapidement le principe du tri par sélection vu en classe de première. Donner les étapes de cet algorithme pour trier la liste `[10,6,3,9,7,5]`
   b. Rappeler rapidement le principe du tri par insertion vu en classe de première. Donner les étapes de cet algorithme pour trier la liste `[10,6,3,9,7,5]`
   c. Quelle est la complexité de ces deux algorithmes ?
2. L'algorithme du **tri fusion** consiste à :

   1️⃣
   partager la liste en deux moitiés (à une unité près),

   2️⃣
   trier chacune des deux moitiés,

   3️⃣
   les fusionner pour obtenir la liste triée.

   On a schématisé le tri de la liste `[10,6,3,9,7,5]` suivant ce principe ci-dessous :

```
graph TD
subgraph Partager en deux
S["[10,6,3,9,7,5]"] --> S1["[10,6,3]"]
S --> S2["[9,7,5]"]
end
subgraph Fusionner
S1 -.Trier.-> T1["[3,6,10]"]
S2 -.Trier.-> T2["[5,7,9]"]
T1 --> T["[3,5,6,7,9,10]"]
T2 --> T
end
```

   a. Le tri des deux moitiés est lui-même effectué par tri fusion, par conséquent que peut-on dire de cet algorithme ?
   b. On a schématisé ci-dessous le fonctionnement complet de l'algorithme pour la liste `[10,6,3,9,7,5]`, recopier et compléter les cases manquantes.

```
graph TD
   subgraph Partager en deux
   S["[10,6,3,9,7,5]"] --> S1["[10,6,3]"]
   S --> S2["[9,7,5]"]
   S1 --> S11["[10]"]
   S1 --> S12["[6,3]"]
```

```
    S2 --> S21["[9]"]
    S2 --> S22["[...,...]"]
    S12 --> S121["[6]"]
    S12 --> S122["[3]"]
    S22 --> S221["[...]"]
    S22 --> S222["[...]"]
  end
    subgraph Fusionner
    S121 --> T21["[...,...]"]
    S122 --> T21
    S221 --> T22["[5,7]"]
    S222 --> T22["[5,7]"]
    S11 --> T1["[...,...,...]"]
    T21 --> T1
    S21 --> T2["[...,...,...]"]
    T22 --> T2
    T1 --> T["[3,5,6,7,9,10]"]
    T2 --> T
  end
```

3. Implémentation en Python

a. Programmer une fonction `partage(l)` qui prend en argument une liste `l` et renvoie les deux moitiés `l1` et `l2` (à une unité près) de `l`. Par exemple `partage([3,7,5])` renvoie `[3]` et `[7,5]`.

> **Aide**
>
> - Penser à utiliser les constructions de listes par compréhension
> - Les *slices* de Python sont un moyen efficace d'effectuer le partage, mais leur connaissance n'est pas un attendu du programme de terminale. Les élèves intéressés pourront faire leur propre recherche sur le *Web*.

b. On donne ci-dessous une fonction `fusion(l1,l2)` qui prend en argument deux listes **déjà triées** `l1` et `l2` et renvoie la liste triée `l` fusion de `l1` et `l2` :

```
1   def fusion(l1,l2):
2       ind1=0
3       ind2=0
4       l = []
5       while ind1<len(l1) and ind2<len(l2):
6           if l1[ind1]<l2[ind2]:
7               l.append(...)
8               ind1+=1
9           else:
10              l.append(...)
11              ind2+=1
12      if ind1==len(l1):
13          for k in range(ind2,len(l2)):
14              l.append(...)
15      else:
16          for k in range(ind1,len(l1)):
17              l.append(...)
18      return l
```

i. Recopier et compléter cette fonction.

ii. Quel est le rôle des variables `ind1` et `ind2` ?

iii. Ajouter un commentaire décrivant le rôle de la boucle `while`.

iv. Ajouter un commentaire décrivant le rôle des lignes 12 à 17.

c. En utilisant les deux fonctions précédentes, écrire une fonction `tri_fusion(l)` qui implémente l'algorithme du tri fusion en Python.

```
def tri_fusion(liste):
    long = len(liste)
    if long <= 1:
        return liste
    else:
        l1, l2 = partage(liste)
        l1 = tri_fusion(l1)
        l2 = tri_fusion(l2)
    return fusion(l1,l2)
```

Important

On montre que l'algorithme du tri fusion a une complexité en $O(n\log(n))$, c'est donc un algorithme plus efficace que le tri par insertion ou le tri par sélection qui ont tous les deux une complexité en $O(n^2)$.

2. Exercices

Maximum des éléments d'une liste

On propose l'algorithme suivant pour la recherche du maximum des éléments d'une liste :

1
Partager la liste en deux moitiés `l1` et `l2`

2
Chercher les maximums `m1` de `l1` et `m2` de `l2`

3
En déduire le maximum `m` de `l`.

1. Expliquer pourquoi cet algorithme fait partie de la méthode **diviser pour régner**.
2. Cet algorithme est-il récursif ? Justifier.
3. Ecrire une implémentation en Python de cet algorithme.

Epreuve Pratique

Le but de l'exercice est de compléter une fonction qui détermine si une valeur est présente dans un tableau de valeurs triées dans l'ordre croissant.

L'algorithme traite le cas du tableau vide.

L'algorithme est écrit pour que la recherche dichotomique ne se fasse que dans le cas où la valeur est comprise entre les valeurs extrêmes du tableau.

On distingue les trois cas qui renvoient `False` en renvoyant `False,1` , `False,2` et `False,3` .

Compléter l'algorithme de dichotomie donné ci-après.

```
1   def dichotomie(tab, x):
2       """
3           tab : tableau trié
4   dans l'ordre croissant
5           x : nombre entier
6           La fonction
7   renvoie True si tab
8   contient x et False
9   sinon
10      """
11      # cas du tableau vide
12      if ...:
13          return False,1
14      # cas où x n'est pas
15  compris entre les
16  valeurs extrêmes
17      if (x < tab[0]) or ...:
18          return False,2
19      debut = 0
20      fin = len(tab) - 1
21      while debut <= fin:
22          m = ...
23          if x == tab[m]:
                return ...
```

```
        if x > tab[m]:
            debut = m + 1
        else:
            fin = ...
    return ...
```

Exemples :

🐍 **Script Python**

```
>>> dichotomie([15, 16, 18, 19, 23, 24, 28, 29, 31, 33],28)
True
>>> dichotomie([15, 16, 18, 19, 23, 24, 28, 29, 31, 33],27)
(False, 3)
>>> dichotomie([15, 16, 18, 19, 23, 24, 28, 29, 31, 33],1)
(False, 2)
>>> dichotomie([],28)
(False, 1)
```

La fonction `fusion` prend deux listes `L1`, `L2` d'entiers triées par ordre croissant et les fusionne en une liste triée `L12` qu'elle renvoie.

Le code Python de la fonction est

```python
def fusion(L1,L2):
    n1 = len(L1)
    n2 = len(L2)
    L12 = [0]*(n1+n2)
    i1 = 0
    i2 = 0
    i = 0
    while i1 < n1 and ... :
        if L1[i1] < L2[i2]:
            L12[i] = ...
            i1 = ...
        else:
            L12[i] = L2[i2]
            i2 = ...
        i += 1
    while i1 < n1:
        L12[i] = ...
        i1 = i1 + 1
        i = ...
    while i2 < n2:
        L12[i] = ...
        i2 = i2 + 1
        i = ...
    return L12
```

Compléter le code.

Exemple :

Script Python

```
>>> fusion([1,6,10],[0,7,8,9])
[0, 1, 6, 7, 8, 9, 10]
```

Soit `T` un tableau non vide d'entiers triés dans l'ordre croissant et `n` un entier.
La fonction `chercher`, donnée à la page suivante, doit renvoyer un indice où la valeur `n` apparaît éventuellement dans `T`, et `None` sinon.

Les paramètres de la fonction sont :

- `T`, le tableau dans lequel s'effectue la recherche ;
- `n`, l'entier à chercher dans le tableau ;
- `i`, l'indice de début de la partie du tableau où s'effectue la recherche ;
- `j`, l'indice de fin de la partie du tableau où s'effectue la recherche.

La fonction `chercher` est une fonction récursive basée sur le principe « diviser pour régner ».

Le code de la fonction commence par vérifier si `0 <= i` et `j < len(T)`.
Si cette condition n'est pas vérifiée, elle affiche `"Erreur"` puis renvoie `None`.

Recopier et compléter le code de la fonction `chercher` proposée ci-dessous :

```
def chercher(T, n, i, j):
    if i < 0 or ??? :
        print("Erreur")
        return None
    if i > j :
        return None
    m = (i + j) // ???
    if T[m] < ??? :
        return chercher(T, n, ??? , ???)
    elif ??? :
        return chercher(T, n, ??? , ??? )
    else :
        return ???
```

L'exécution du code doit donner :

🐍 Script Python

```
>>> chercher([1,5,6,6,9,12],7,0,10)
Erreur
>>> chercher([1,5,6,6,9,12],7,0,5)
>>> chercher([1,5,6,6,9,12],9,0,5)
4
>>> chercher([1,5,6,6,9,12],6,0,5)
2
```