INFORMATIQUE, CORRIGÉ DU DEVOIR LIBRE N^O 1

À rendre le lundi 7 janvier 2014

▷ *Le candidat attachera la plus grande importance à la clarté, à la précision et à la concision de la rédaction.*

▷ *Si le candidat découvre en cours d'épreuve ce qu'il croît être une erreur d'énoncé, il le précisera dans sa copie.*

▷ *Le sujet comporte un problème sur le calcul des éléments dominants et l'élément majoritaire d'une liste d'entiers naturels.*

1. ÉLÉMENT(S) DOMINANT(S) ET ÉLÉMENT MAJORITAIRE D'UNE LISTE

▷ On étudie dans ce problème des algorithmes classiques de recherche des éléments dominants et de l'élément majoritaire d'une liste.

▷ Toute les listes manipulées dans ce problème seront supposées à valeurs dans $\mathbb{N}$.

▷ Soient `t` une liste de longueur `n` et `x` un élément de `t`. On appelle fréquence de `x` dans `t` le nombre d'éléments de l'ensemble

$$\{i \in [\![0, n-1]\!] \,;\, t[i] = x\}$$

PARTIE 1 – ÉLÉMENT(S) DOMINANT(S) D'UNE LISTE

On appelle élément dominant d'une liste `t` tout élément de `t` de fréquence maximale. Toute liste non vide admet un élément dominant mais celui-ci n'est pas toujours unique. Considérons par exemple, les listes :

`t1=[5,1,5,4,5,7,5,3,5]` et `t2=[1,4,4,5,7,2,3,5]`

La liste `t1` admet un unique élément dominant, l'entier `5`. La liste `t2` admet deux éléments dominants, les entiers `4` et `5`. On étudie dans cette partie des algorithmes calculant la liste des éléments dominants d'une liste.

a) ***Un premier algorithme.***

i) Écrire une fonction `maxListe(u)` renvoyant le plus grand élément d'une liste `u`. On donnera un invariant de boucle et on démontrera que la fonction renvoie bien le résultat attendu.

ii) Écrire une fonction `listeFrequences(t)` renvoyant la liste `freq` de même longueur `n` que `t` et telle que, pour tout `i`$\in [\![0, n-1]\!]$,

$$\texttt{freq[i]} = \begin{cases} \bullet \text{ la fréquence de } \texttt{t[i]} \text{ dans } \texttt{t}, \text{si } \texttt{i} \text{ est} \\ \quad \text{le plus petit indice } \texttt{j} \text{ tel que } \texttt{t[j]=t[i]} \text{ ;} \\ \bullet \text{ 0, sinon.} \end{cases}$$

Par exemple, pour les listes `t1` et `t2`, la fonction `listeFrequences` doit renvoyer les tableaux suivants :

`[5,1,0,1,0,1,0,1,0]` et `[1,2,0,2,1,1,1,0]`

iii) Écrire une fonction `elementsDominants(t)` renvoyant la liste des éléments dominants d'une liste `t` et utilisant les fonctions `listeFrequences(t)` et `maxListe(u)`.

iv) Justifier que la fonction `elementsDominants` s'exécute en temps quadratique.

b) ***Un second algorithme.***

i) Écrire une fonction `elementsDominantsTri(t)` renvoyant la liste des éléments dominants d'une liste triée par ordre croissant `t`, ie telle que

$$\texttt{t[0]} \leqslant \texttt{t[1]} \leqslant \cdots \leqslant \texttt{t[n-1]}$$

On donnera (avec justification) un algorithme en temps linéaire.

ii) On calcule les éléments dominants d'une liste `t` en triant `t` puis en appliquant la fonction précédente. Quel est l'ordre de grandeur du temps d'exécution ? On admettra que l'algorithme de tri s'exécute en un temps dominé par $n\ell\text{n}(n)$.

PARTIE 2 – ÉLÉMENT MAJORITAIRE

Un élément `x` d'une liste `t` est dit majoritaire si sa fréquence dans `t` est strictement supérieure à `n/2`, où `n` est la longueur de la liste. Il est clair qu'une liste admet au plus un élément majoritaire. Considérons par exemple, les listes :

`t1=[5,1,5,4,5,7,5,3,5]` et `t2=[1,4,4,5,7,2,3,5]`

La liste `t1` admet un élément majoritaire, l'entier `5`. En revanche, la liste `t2` n'admet aucun terme majoritaire.

a) ***L'algorithme naïf.***
Un algorithme élémentaire de détermination de l'élément majoritaire d'une liste `t` de longueur `n` consiste à calculer le nombre d'occurrence(s) d'un élément de `t` (au moyen d'une boucle), tant que celui-ci ne dépasse pas `n/2` et tant qu'il reste des éléments à tester.

i) Écrire une fonction `elementMajoritaire(t)` renvoyant `-1` si `t` n'admet aucun élément majoritaire, et renvoyant l'élément majoritaire de `t` en cas d'existence.

ii) Quel est l'ordre de grandeur du temps d'exécution de cette fonction ?

b) ***Amélioration passant par un tri.***

i) Écrire une fonction `elementMajoritaireTri(t)` agissant comme la fonction `elementMajoritaire` mais sur une liste `t` supposée triée par ordre croissant.

ii) On calcule l'élément majoritaire d'une liste `t` en triant `t` puis en appliquant la fonction précédente. Quel est l'ordre de grandeur du temps d'exécution ? On admettra que l'algorithme de tri s'exécute en un temps dominé par $n\ell n(n)$.

PARTIE 3 – ÉLÉMENT MAJORITAIRE : L'ALGORITHME DE MOORE

Soit `t` une liste de longueur `n`.

a) On considère l'algorithme suivant :

Algorithme 1 : Algorithme de Moore (1980)

```
Initialisation : n=len(t), c=0, x=t[0];
pour 0 ⩽ i ⩽ n-1 faire
    si c==0 alors
        x=t[i];
        c=1;
    sinon
        si t[i]==x alors
            c=c+1;
        sinon
            c=c-1;
si c==0 alors
    Renoyer None;
sinon
    Renvoyer x;
```

i) Montrer que la propriété suivante

$$\begin{cases} [\![0, i-1]\!] \text{ est la réunion de deux parties disjointes } \mathscr{C} \text{ et } \mathscr{P} \text{ telles que :} \\ \bullet\ \#\mathscr{C} = \texttt{c}\,; \\ \bullet \text{ pour tout } j \in \mathscr{C},\ \texttt{t[j]=x}\,; \\ \bullet \text{ les éléments de la sous-liste } \texttt{[t[j] for j in } \mathscr{P}\texttt{]} \text{ peuvent être regroupés en paires } \texttt{[u,v]} \text{ avec } \texttt{u}\neq\texttt{v}. \end{cases}$$

est un invariant de boucle de cet algorithme pour $i \geqslant 1$.

ii) Montrer que si `t` admet un élément majoritaire `m`, alors l'algorithme renvoie `m`. On pourra raisonner par l'absurde.

iii) On suppose que l'algorithme ne renvoie pas `None` mais un entier `m`. L'élément `m` est-il nécessairement majoritaire dans `t` ?

b) En déduire une fonction `elementMajoritaireMoore(t)` agissant comme `elementMajoritaire(t)` mais utilisant l'algorithme de Moore.

c) Quel est le temps d'exécution de cette fonction ? On justifiera avec soin l'ordre de grandeur obtenu.

Corrigé

1. Élément(s) dominant(s) et élément majoritaire d'une liste

Partie 1 – Élément(s) dominant(s) d'une liste

a) ***Un premier algorithme.***

i) Voici un algorithme itératif reposant sur une boucle inconditionnelle.

```
def maxListe(t):
    maxi,n=t[0],len(t)
    for i in range(1,n):
        if t[i]>maxi:
            maxi=t[i]
    return(maxi)
```

La propriété suivante :

`maxi` est le maximum de `t[0],...,t[i]`

est clairement un invariant de boucle. Comme en fin de boucle `i=n-1`, on en déduit que `maxi` est alors égal au maximum de `t` et que l'algorithme est correct.

ii) Pour chaque terme de la liste, on commence par repérer son premier rang d'apparition, puis on calcule sa fréquence et on remplit le tableau des fréquences.

```
def listeFrequences(t):
    n=len(t)
    aux=[0]*n
    for i in range(n):
        imin,freq=0,1
        while t[imin]!=t[i]:
            imin=imin+1
        for j in range(imin+1,n):
            if t[j]==t[i]:
                freq=freq+1
        aux[imin]=freq
    return(aux)
```

iii) On calcule la fréquence maximale de `listeFrequences(t)` puis on crée la liste des éléments ayant cette fréquence, ce sont les éléments dominants de la liste.

```
def elementsDominants(t):
    n=len(t)
    frequences=listeFrequences(t)
    Nmax=maxListe(frequences)
    dominants=[]
    for i in range(n):
```

```
        if frequences[i]==Nmax:
            dominants.append(t[i])
    return(dominants)
```

iv) La boucle de `elementsDominants(t)` s'exécute en temps linéaire tout comme l'appel `maxListe(t)`. Étudions le temps d'exécution de l'appel `listeFrequences(t)` : pour chaque valeur de `i` dans $[\![0, n-1]\!]$, la seconde boucle `for` effectue exactement `n-imin-1` itétrations. Le temps d'exécution de `listeFrequences(t)` est donc quadratique. Au total, l'appel `elementsDominants(t)` s'exécute en un temps quadratique dans le pire des cas.

b) ***Un second algorithme.***

i) On calcule d'abord la longueur maximale d'un palier de `t` (ie une séquence de termes égaux) puis on crée la liste des termes correspondants, ce sont les éléments dominants de `t`.

```
def elementsDominantsTri(t):
    n=len(t)
    i,max=0,0
    while i<n-1:
        j=i+1
        compteur=0
        while j<n-1 and t[i]==t[j]:
            compteur=compteur+1
            j=j+1
        i=j
        if compteur>max:
            max=compteur
    dominants=[]
    for i in range(n-max):
        if t[i+max]==t[i]:
            dominants.append(t[i])
    return(dominants)
```

Comme la liste n'est parcourue que deux fois, le temps d'exécution est linéaire.

ii) Comme $n = o(n\ell n(n))$, la fonction `elementsdominantsTri` s'exécute en un temps dominé par $n\ell n(n)$.

Partie 2 – Élément majoritaire

a) ***L'algorithme naïf.***

i) On effectue une double boucle `while` pour calculer le nombres d'occurrences `c` de `t[i]` dans `[t[i],...,t[n-1]]`. Dès que `c>n//2`, on est certain que `t[i]` est majoritaire dans `t`.

```
def elementMajoritaire(t):
    n,i=len(t),0
    while i<n:
```

```
        c,j=0,i
        while j<n:
            if t[j]==t[i]:
                c=c+1
                if c>n//2:
                    return(t[i])
            j=j+1
        i=i+1
    return(-1)
```

ii) Comme l'algorithme consiste en deux boucles `while` imbriquées, son temps d'exécution est au maximum quadratique. L'exemple d'une liste dont les termes sont deux à deux distincts montre que le temps d'exécution est exactement quadratique dans le pire des cas.

b) ***Amélioration passant par un tri.***

i) Il suffit d'étudier l'existence d'un palier de longueur $n/2+1$ dans la liste `t` pour conclure à l'existence ou non d'un élément majoritaire dans `t`.

```
def elementMajoritaireTri(t):
    n=len(t)
    m=n//2
    for i in range(n-m):
        if t[i]==t[i+m]:
            return(t[i])
    return(-1)
```

Comme la liste n'est parcourue qu'une seule fois, le temps d'exécution est linéaire.

ii) Comme $n = o(n\ell n(n))$, la fonction `elementMajoritaire` s'exécute en un temps dominé par $n\ell n(n)$.

PARTIE 3 – ÉLÉMENT MAJORITAIRE : L'ALGORITHME DE MOORE

a) Étude théorique de l'algorithme de Moore.

i) Montrons que la propriété suivante

$$\left\{\begin{array}{l}[\![0,i-1]\!] \text{ est la réunion de deux parties disjointes } \mathscr{C} \text{ et } \mathscr{P} \text{ telles que :}\\ \bullet\ \#\mathscr{C} = \texttt{c};\\ \bullet\ \text{pour tout } j\in\mathscr{C},\ \texttt{t[j]=x};\\ \bullet\ \text{les éléments de la sous-liste } \texttt{[t[j] for j in } \mathscr{P}\texttt{]} \text{ peuvent être regroupés en paires } \texttt{[u,v]} \text{ avec } \texttt{u}\neq\texttt{v}.\end{array}\right.$$

est un invariant de boucle de cet algorithme pour $i \geqslant 1$.

- Au début du tour de boucle où `i=1`, on a `c=1` et `x=t[0]`. En posant $\mathscr{P} = \emptyset$ et $\mathscr{C} = \{0\}$, on a clairement deux ensemble vérifiant la propriété.
- Supposons la propriété vérifiée au début du tour de boucle correpondant à `i`. Il y a trois cas à considérer :

  ▷ Si `c=0`, `c` devient `c'=1` et `x=t[i]`. La propriété est clairement vérifiée en fin d'itération, il suffit de considérer $\mathscr{C}' = \{i\}$ et $\mathscr{P}' = \mathscr{P}$.

▷ Si `c≠0` et `x=t[i]`, `c` devient `c'=c+1`. La propriété est clairement vérifiée en fin d'itération, il suffit de considérer $\mathscr{C}' = \mathscr{C} \cup \{i\}$ et $\mathscr{P}' = \mathscr{P}$.

▷ Si `c≠0` et `x≠t[i]`, `c` devient `c'=c-1`. On note $i_0 = \inf(\mathscr{C})$ (qui est bien défini car `c`$= \#\mathscr{C} > 0$), on pose $\mathscr{C}' = \mathscr{C} \setminus \{i_0\}$ et $\mathscr{P}' = \mathscr{P} \cup \{i_0, i\}$. On a

`[t[j] for j in `$\mathscr{P}'$`]=[t[j] for j in `$\mathscr{P}$`,t[`i_0`],t[i]]`

Comme `x=t[`i_0`]≠t[i]` et puisque les éléments de la sous-liste `[t[j] for j in `$\mathscr{P}$`]` peuvent être regroupés en paires `[u,v]` avec `u≠v`, il en est de même pour `[t[j] for j in `$\mathscr{P}'$`]`. La propriété reste donc vérifiée en fin d'itération.

On en déduit que la propriété est un invariant de boucle.

ii) Raisonnons par contraposition. Supposons que l'algorithme de Moore renvoie `None`. Dans ce cas `c=0` en fin de boucle, et d'après l'invariant prouvé à la question 3) a) i), les éléments de `t` sont regroupables en paires `[u,v]` avec `u≠v`. Ainsi aucun élément de `t` n'apparaît strictement plus de $n/2$ fois dans `t` et donc `t` n'admet pas d'élément majoritaire.

iii) Si l'algorithme ne renvoie pas `None` mais un entier `m`, on ne peut en conclure que l'élément `m` est majoritaire dans `t`, comme le prouve le contre-exemple suivant : `t=[1,2,3]`.

b) Dans le cas où l'algorithme renvoie `x`, il n'est pas certain que cet élément soit majoritaire, on effectue donc une vérification avant de conclure.

```
def elementMajoritaireMoore(t):
    n,c,x=len(t),0,t[0]
    for i in range(n):
        if c==0:
            x=t[i]
            c=1
        else:
            if t[i]==x:
                c=c+1
            else:
                c=c-1
    if c==0:
        return(-1)
    else:
        compt=0
        for i in range(n):
            if t[i]==x:
                compt=compt+1
        if compt>n//2:
            return(x)
        else:
            return(-1)
```

c) Comme la liste n'est parcourue que deux fois, le temps d'exécution est linéaire.