teaching sciences for innovation

Proposition de corrigé

Concours : Concours Commun Mines-Ponts

Année : 2016

Filière : MP - PC - PSI

Épreuve : Informatique

Ceci est une proposition de corrigé des concours de CPGE, réalisée bénévolement par des enseignants de Sciences Industrielles de l'Ingénieur et d'Informatique, membres de l'UPSTI (Union des Professeurs de Sciences et Techniques Industrielles), et publiée sur le site de l'association :

https://www.upsti.fr/espace-etudiants/annales-de-concours

A l'attention des étudiants

Ce document vous apportera des éléments de corrections pour le sujet traité, mais n'est ni un corrigé officiel du concours, ni un corrigé détaillé ou exhaustif de l'épreuve en question.

L'UPSTI ne répondra pas directement aux questions que peuvent soulever ces corrigés : nous vous invitons à vous rapprocher de vos enseignants si vous souhaitez des compléments d'information, et à vous adresser à eux pour nous faire remonter vos éventuelles remarques.

Licence et Copyright

Informez-vous !

Retrouvez plus d'information sur les Sciences de l'Ingénieur, l'orientation, les Grandes Ecoles ainsi que sur les Olympiades de Sciences de l'Ingénieur et sur les Sciences de l'Ingénieur au Féminin sur notre site : www.upsti.fr

L'équipe UPSTI

Concours Mines-Ponts 2016 – Toutes Filières
Épreuve d'Informatique - Correction

Q1 : C'est un tri par insertion
Après l'itération n° :

```
1  :  L=[ 2 , 5 , 3 , 1 , 4 ]
2  :  L=[ 2 , 3 , 5 , 1 , 4 ]
3  :  L=[ 1, 2 , 3 ,  5 , 4 ]
3  :  L=[ 1, 2 , 3 ,  4 , 5 ]
```

Q2 :
Soit P_i la propriété *« la liste L[0:i+1] est triée dans l'ordre croissant à l'issue de l'itération i »*
Montrons que cette propriété constitue un invariant de boucle :

Initialisation :
P_0 est vraie car la liste L[0:1] ne contient que le premier élément de la liste L, et est donc forcément triée.
Ou bien , si l'on considère que la liste ne peut être vide :
A l'issue de l'itération 1, on s'est occupé de la sous liste de L ne contenant que les deux premiers élément :

- Si L[1]<L[0] alors on a échangé L[1] et L[0] (aux lignes 7 et 8)
- Sinon, on n'a pas touché à la liste

P_1 est donc vraie

Hérédité :
On suppose P_i vraie : *« la liste L[0:i+1] est triée dans l'ordre croissant à l'issue de l'itération i »*
On a donc :

$$L[0] \leq L[1] \leq \ldots\ldots = < L[i]$$

- 1er cas : $L[i] \leq L[i+1]$: la liste L[0:i+2] est triée, P_{i+1} est donc vraie
- 2nd cas : $L[i] > L[i+1]$

Il existe donc un entier k strictement positif tel que $k \leq i$ et $L[k-1] \leq L[i+1] < L[k]$

On est à l'itération *i+1, l*e programme évolue comme suit :

ligne 4 : $j = i+1$
ligne 5 : $x = L[i+1]$: *on « sauvegarde » la valeur à insérer à la bonne place*

On entre dans la boucle *while* car la condition associée est vraie.
La liste est parcourue vers la gauche depuis l'indice i et chaque élément est décalé vers la droite à la ligne 7, puis j est décrémenté de 1.
La boucle *while* est donc parcourue *i-k+1*fois jusqu'à ce que j=k (et donc que la condition $x < L[j-1]$ ne soit plus vérifiée), ou que l'on ait j=0, si l'élément L[i+1] est le plus petit de la sous liste L[0:i+2].

La sortie de boucle est garantie grâce au variant j, strictement positif à l'entrée de la boucle *while,* et qui décroît à chaque itération.

A la sortie de la boucle, l'élément L[i+1] de la liste L est inséré à la position k par l'instruction ligne 9. Les termes supérieurs à cet élément ont été décalé d'un rang vers la droite :
la liste L[0:i+2] est triée, P_{i+1} est donc vraie.

Correction en sortie :
A la sortie de l'algorithme, on a effectué la n-1^{ième} itération et l'on est donc allé jusqu'au bout de la liste : La liste L[0:n] est donc triée.

Q3 :

- Complexité dans le meilleur des cas : *O(n)*

La boucle *while* n'est jamais exécutée (ce qui signifie que la liste est déjà triée dans l'ordre croissant au départ), il y a alors n exécutions de la boucle for .

- Complexité dans le pire des cas : $O(n^2)$

La boucle *while* est toujours exécutée jusqu'à ce qu'on ait j=0 (cela correspond à une liste triée à l'envers). On a alors i opérations dans la boucle *while* et $\sum_{i=1}^{i=n}$ opérations au total. Soit $\frac{n(n-1)}{2}$ opérations .

Le **tri par fusion** est plus efficace dans le pire des cas. Sa complexité dans le pire comme dans le meilleur des cas est en *O(n.ln(n))*.

Q4 :

```
def tri_chaine(L):
    n=len(L)
    for i in range(1,n):
        j=i
        x=L[i]
        while 0<j and x[1]<L[j-1][1]:
            L[j]=L[j-1]
            j=j-1
        L[j]=x
```

Q5 : Aucun champ ne peut servir de clé primaire car aucun d'eux n'est suffisant pour différencier les enregistrements. En revanche les couples {nom,année} ou {iso, année} peuvent servir de clé primaire (dans le mesure où l'on considère qu' « une clé primaire peut posséder plusieurs attributs »)

Q6 :

```
SELECT * FROM palu WHERE annee=2010 AND deces>=1000 ;
```

Q7 :

```
SELECT  nom, cas*100000/pop
 FROM  palu JOIN demographie ON iso=pays AND annee=periode WHERE annee=2011
```

Q8 :

```
SELECT nom FROM palu WHERE annee = 2010 AND cas=(SELECT max(cas) FROM palu WHERE
cas!= (SELECT MAX(cas) FROM palu WHERE annee=2010));
```

Q9 :

```
tri_chaine(deces2010)
```

Q10 :

$X=(S,I,R,D)$ et $f:X\rightarrow(-r.S.I, r.S.I-(a+b).I, a.I, b.I)$

La fonction f est définie sur R^4

Q11 :

```
def f(X) :
      """ Fonction définissant l'équation différentielle """
      global r,a,b
      return np.array([-r*X[0]*X[1],r*X[0]*X[1]-(a+b)*X[1],a*X[1],b*X[1]])
```

Q12 :

La solution avec 7 points de calcul montre des oscillations pour les quantités I et R, que ne montre pas la solution avec 250 points. Ce sont des oscillations du schéma numérique, car le nombre de points n'est pas assez important (pas de temps de calcul trop grand).

La simulation avec 250 points a nécessairement pris plus de temps, mais la précision semble meilleure (il conviendrait néanmoins de comparer avec des solutions exactes si elles existent, ou avec des résultats de mesure)

Q13 :

```
7 |   return np.array([-r*X[0]*Itau,r*X[0]*Itau-(a+b)*X[1],a*X[1],b*X[1]])
...
28|   if i < p  :
29|         Itau = 0.05
30|   else :
31|         Itau = XX[i-p][1]
32|   X = X + df * f(X,Itau)
33|
```

Q14 :

On modifie la valeur de Itau dans le bloc #méthode d'Euler, dans le cas ou $t>\tau$. on insère donc ligne 31 les 4 lignes suivantes

```
31|         s = 0
32|         for j in range(p) :
33|               s = s + XX[i-j][1]*h(j*dt)
34|         Itau = s*dt
```

Q15 :

La fonction grille retourne une matrice carrée de coté n, sous forme de liste de listes, remplie de zéros.

Q16 :

```
def init(n):
      G=grille(n)
      i=randrange(n)
      j=randrange(n)
      G[i][j]=1
      return G
```

Q17 :

```
def compte(G):
    n = len(G)
    nC = [0,0,0,0]
    for i in range(n):
        for j in range(n):
            nC[G[i][j]] += 1
    return nC
```

Q18 :

La fonction renvoie un booléen : vrai si la case est exposée, faux sinon.

Q19 :

```
11 |   elif i == 0:
12 |     return (G[0][j-1]-1)*(G[0][j+1]-1)*(G[1][j-1]-1)*(G[1][j]-1)*(G[1][j+1]-1) == 0
```

....

```
19 |   else:
20 |      return (G[i-1][j-1]-1)*(G[i-1][j]-1)*(G[i-1][j+1]-1)*(G[i][j-1]-1)*\
21 |      (G[i][j+1]-1)*(G[i+1][j-1]-1)*(G[i+1][j]-1)*(G[i+1][j+1]-1) == 0
```

Q20 :

```
def suivant(G,p1,p2):
      n=len(G)
      H=grille(n)    #nouvelle grille
      for i in range(n):
            for j in range(n):
                  if G[i][j]==0 and est_exposee(G,i,j) and bernouilli(p2)==1: #si casse saine
                        H[i][j]=1    #elle devient infectée avec probabilité p2 si exposée
                  elif G[i][j]==1:              #si la case est infectée
                        if bernouilli(p1)==1:
                              H[i][j]=3
                        else :
                              H[i][j]=2
                  elif G[i][j]==2 or G[i][j]==3:       #si la casse est décédée ou rétablie
                        H[i][j]=G[i][j]             #rien ne change
      return H
```

Q21 :

```
def simulation(n,p1,p2):
    G=init(n)
    H=suivant(G,p1,p2)
    while G!=H:
        G=H
        H=suivant(G,p1,p2)
    resu=compte(H)
    return [resu[i]/n**2 for i in range(4)]
```

Q22 :

En fin de simulation, il ne reste plus de case infectée : $x_1=0$

On a $x_0+x_2+x_3=1$ et $x_{atteinte}=x_2+x_3=1-x_0$

Q23 :

```
def seuil(Lp2,Lxa):
    n=len(Lp2)
    g,d=0,n-1
    while d!=g+1 :
        m=(g+d)//2
        if Lxa[m]==0.5:
            return [Lp2[m],Lp2[m]]
        elif Lxa[m]>0.5:
            d=m
        else:
            g=m
    return [Lp2[g],Lp2[d]]
```

Remarque : Il y a un problème avec l'énoncé : le cas initial doit être exclu si l'on veut obtenir la courbe de la figure 2 (0 en 0)

Q24 :

On souhaite étudier l'influence d'une campagne de vaccination dans le cas où il y a un risque de propagation.

Si l'on supprime la ligne 8, le risque est que la case infectée issu de l'appel de la fonction init() soit vaccinée, et que le cas passe à rétabli. Il n'y aura tout simplement pas de possibilité de simuler la propagation de l'épidémie.

Q25 :

L'appel *init_vac(5,0.2)* renvoie une grille 5*5 avec 1 case infectée et 5 cases vaccinées (ou seulement 4 si l'on a tenté de vacciner la case infectée) et 19 cases saines.