Éléments de correction : Mines 2016 – IPT (toutes filières)

1 Tri et bases de données

Q1. On obtient successivement
$n = 5$
$i = 1$ $x = 2$ $[2, 5, 3, 1, 4]$
$i = 2$ $x = 3$ $[2, 3, 5, 1, 4]$
$i = 3$ $x = 1$ $[1, 2, 3, 5, 4]$
$i = 4$ $x = 4$ $[1, 2, 3, 4, 5]$

Q2. Notons $P(k)$:" à l'issue de l'itération pour la valeur $i = k$, la valeur L_k de la variable L est telle que $L_k[0 : k+1]$ est la liste initiale $L_0[0 : k+1]$ triée dans l'ordre croissant, et $L_k[k+1 :] = L_0[k+1 :]$."
• La liste $L_0[0 : 0+1]$ est la liste ayant une seule valeur $L_0[0]$ donc $P(0)$ est vrai (à l'issue de l'itération pour $i = 0$ signifiant avant d'entrer dans l'itération pour $i = 1$ i.e. la première).
• Supposons $P(k)$ vrai et $k + 1 \leqslant n$ (i.e. on fait encore une itération).
Comme j prend initialement la valeur $k + 1$ et ne peut que décroître, et comme on ne modifie que la valeur de $L[j]$, on aura bien $L_{k+1}[k + 2 :] = L_k[k + 2 :] = L_0[k + 2 :]$ (via $P(k)$).
On a aussi $L_k[0 : k + 1] = [x_0, \dots x_k]$ avec $x_0 \leqslant \cdots \leqslant x_k$. Au début de l'itération j vaut $k + 1$ et $x = L_k[k + 1] = L_0[k + 1]$. La boucle "while" permet de faire décroître j et de s'arrêter dès que $L_k[j] \leqslant x$. Quand on s'arrête, on a donc dans $L[0 : k + 1]$ la valeur $[x_0, \cdots x_j, x_j, \cdots x_k]$, puis la ligne 9 assure $L_{k+1}[0 : k + 1] = [x_0, \cdots x_{j-1}, x, x_j, \cdots x_k]$ i.e. c'est bien $L_k[0 : k] + [x] = L_k[0 : k + 1]$ trié par ordre croissant et via $P(k)$, c'est bien aussi $L_0[0 : k + 1]$ trié par ordre croissant.
• Ainsi $P(k)$ est bien un invariant de boucle.

A la sortie du programme, L est à la valeur L_{n-1} (la dernière valeur prise par i est $n - 1$ donc via $P(n - 1)$, L contient bien la liste initiale triée par ordre croissant.

Q3. La ligne 1 compte 2 opérations élémentaires (une affectation et un appel à la fonction len). *On pourrait compter $n + 1$ si on pense que* len *est plutôt de complexité linéaire en la longueur de la liste passée en paramètre.*
La variable i prend toutes les valeurs de 1 à $n - 1$ et pour une valeur i fixée, on a : - (ligne 4) une opération élémentaire (affectation)
- (ligne 5) deux opérations élémentaires (affectation, appel à un élément d'une liste)
- (lignes 6-7-8) : 4 opérations élémentaires dans le test (2 comparaisons, une opération booléenne et un appel à un élément d'une liste) et pour chaque passage dans la boucle (dans le meilleur des cas : aucun, dans le pire i) il y a 3+2=5 opérations élémentaires. - (ligne 9) une opération élémentaire (affectation)

Finalement, dans le meilleur des cas (liste déjà triée dans l'ordre croissant), on fait de l'ordre de $2 + \sum_{i=1}^{n-1}(1 + 2 + 4 + 0 + 1) = 8n - 5 = O(n)$ opérations élémentaires d'où une complexité linéaire ;

dans le pire des cas (liste triée dans l'ordre décroissant), on fait $2 + \sum_{i=1}^{n-1}(1 + 2 + 4 + 5i + 1)$ i.e. $8n - 5 + \frac{5(n-1)n}{2} = O(n^2)$ opérations élémentaires d'où une complexité quadratique.

Dans le pire comme dans le meilleur des cas, le tri fusion est plus efficace car de complexité logarithmique en $O(\ln(n))$.

Q4. À partir de tri, on peut proposer

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

Q5. Une clé primaire est un ensemble minimal d'attributs dont la valeur définit de manière unique tout enregistrement.
Ainsi aucune attribut seul n'est une clé primaire pour la table palu : un même pays (nom ou iso) apparaissant dans plusieurs enregistrements (années différentes), pour une même année plusieurs pays apparaissant dans palu,..etc.
En revanche, le couple (annee,nom) (ou (annee,iso)) est une clé primaire.

Q6. On peut écrire

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

Q7. On peut écrire (on garde pays pour savoir de quel pays il s'agit même si ce n'est pas explicitement demandé)

```
SELECT pays, 100 000 *cas/pop as taux incidence
    FROM palu JOIN demographie ON pays=iso AND periode=annee
    WHERE annee=2011;
```

Q8. Pour l'année 2010, on cherche le plus grand nombre de nouveaux cas de paludisme, on crée une nouvelle table virtuelle sans ce dernier et on cherche le pays ayant eu le maximum de cas (restants)...

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

Q9. D'après la requête utilisée, deces201 contient la liste des couples (nom du pays, nombre de décès en 2010). Ainsi, on peut trié par ordre croissant du nombre de décès par

```
tri_chaine(deces2010)
```

Remarquons qu'avant la conversion python, on aurait pu faire directement trier le résultat via la base de données....

2 Modèle à compartiments

Q10. On peut prendre $X = (S, I, R, D)$ et $f : (x; y; z; w) \in \mathbb{R}_+^4 \mapsto (-rxy; rxy - (a + b)y; ay; by)$.

Q11. On remplace la ligne 4 (indentée comme dans l'énoncé) par

```
return np.array([-r X[0]*X[1];r X[0]*X[1]-(a+b)*X[1];a*X[1];b*X[1]])
```

Q12. Plus N est grand, plus le temps de calcul de simulation est grand (plus grand nombre de valeurs à calculer) mais plus le pas dt est petit donc meilleure est l'approximation (On peut assimiler une courbe avec sa tangente mais localement !). La simulation avec N=250 fournit beaucoup plus de points

d'où un tracé qui semble continu à l'oeil (deux points successifs pour deux valeurs successives de t sont trop proches pour qu'on les distingue facilement visuellement). Cette simulation est aussi plus proche de la solution réelle.

Q13. On remplace la ligne 7 (indentée comme dans l'énoncé) par

```
return np.array([-r X[0]*Itau;r X[0]*Itau-(a+b)*X[1];a*X[1];b*X[1]])
```

et on place en ligne 28

```
if i<p :
   Iretard=X0[1]
else:
   Iretard=XX[i-p][1]
X=X+dt*f(X,Iretard)
```

Pour la dernière ligne, la syntaxe correspondant au fait que numpy *travaille en "vectorialisé".*

Q14. On est comme dans les deux cas précédents mais dans le premier, on multiplie par $I(t)$, dans le deuxième par $I(t-\tau)$ et ici par $\int_0^\tau I(t-s)\,h(s)\mathrm{d}s$. Donc on reprend le code de la question Q13 (en particulier la ligne 7) mais à la place de ce qui a été proposé en ligne 28, on peut mettre

```
mult=0
for j in range(p):
        if j<i+p : # i.e. t_i-t_j<Tau
           mult=mult+h(j*dt) * X0[1]
    else:
       mult=mult+h(j*dt) * XX[j-i][1]
X=X+dt*f(X,mult)
```

3 Modélisation dans des grilles

Q15. La boucle en j fabrique une liste L à n éléments tous nuls, et celle en i une liste M à n éléments tous égaux à la liste précédente. Donc au sens de l'énoncé, grille(n) renvoie une grille de taille $n \times n$ remplie de 0.

Q16. On peut écrire

```
def init(n):
   G=grille(n)
   G[rd.randrange(n)][rd.randrange(n)]=1
   return G
```

Q17. On parcourt toute la grille et on compte le nombre de cases dans chaque état (nb est la liste qui en indice i contient le nombre de cases visitées dans l'état i) :

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

Q18. La fonction est_exposee renvoie un booléen.

Q19. On rappelle qu'un produit de $G[i][j]-1$ est nul si et seulement l'un d'un terme $G[i][j]$ vaut 1 ; cela permet donc de tester si une case a au moins une case voisine infectée.
Quand on exécute la ligne 11, c'est que i vaut 0 et j ne vaut ni 0 ni $n-1$, autrement dit on est sur une case sur le bord mais pas un coin, donc on peut mettre en ligne 12

```
produit=(G[0][j-1]-1)*(G[0][j+1]-1)
for k in range(3):
   produit= produit*(G[1][j-1+k]-1)
return produit
```

Quand on exécute la ligne 20, c'est que ni i ni j ne valent 0 ou $n-1$, autrement dit on est sur une case $[i,j]$ qui n'est pas sur un bord de la grille, donc on peut mettre en ligne 20

```
produit=(G[i][j-1]-1)*(G[i][j+1]-1)
for k in range(3):
   produit= produit*(G[i-1][j-1+k]-1)*(G[i+1][j-1+k]-1)
return produit
```

Q20. On remplit une nouvelle grille : on a besoin de la grille à l'instant précédent pour déterminer la proximité de cases infectées ! Il est donc impossible de faire évoluer la grille directement in situ.

```
def suivant(G,p1,p2):
   n=len(G)
   nouv=grille(n)
   for i in range(n):
      for j in range(n):
        if G[i][j]>1:
           nouv[i][j]=G[i][j]
        elif G[i][j]==1:
           if bernoulli(p1)==1:
               nouv[i][j]=3
           else:
               nouv[i][j]=2
        else:
            if est_exposee(G,i,j):
               if bernoulli(p2)==1:
                  nouv[i][j]=1
   return nouv
```

Q21. On s'arrête quand la grille ne peut plus évoluer

```
def simulation(n,p1,p2):
   G=init(n)
   nb=compte(G)
   while nb[1]>0:
      G=suivant(G,p1,p2)
      nb=compte(G)
   return [nb[0]/n*n,nb[1]/n*n,nb[2]/n*n,nb[3]/n*n]
```

Q22. On remarque que la valeur de x_1 est connue, ce sera forcément 0 puisque la grille n'évolue plus. On a aussi $x_0+x_1+x_2+x_3=n^2$ le nombre de cases dans la grille.
Comme on part d'une grille avec que des sains et des infectés, toute "case" ayant été à un moment ou un autre dans l'état infecté (1), est à la fin de la simulation dans l'état rétabli (2) ou décédé (3), donc on a $x_{\text{atteinte}} = (x_2+x_3)/n^2 = 1 - x_0/n^2$