

## Sujet : Informatique toutes filières Mines Ponts 2016

### MODÉLISATION DE LA PROPAGATION D'UNE ÉPIDÉMIE

D'après le corrigé proposé par E. Méthou (U.P.S.)

#### Première partie

### Tri et bases de données

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

**Q 1.**  $n = 5$

Voici le contenu de la liste  $L$  à la fin de chaque itération :

( $i = 1, x = 2$ ) : [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]

**Q 2.** *Invariant de boucle*

La question posée ainsi appelle une démonstration par récurrence, que vous pouvez, c'est votre choix, bâcler ou passer...

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  $L_k[k + 1 :] = L_0[k + 1 :]$ ."

— **Initialisation** : La liste  $L_0[0 : k + 1]$  est la liste ayant une seule valeur  $L_0[0]$  donc  $P(0)$  est vraie (à l'issue de l'itération pour  $i = 0$  signifiant avant d'entrer dans l'itération pour  $i = 1$ , i.e. la première).

— **Hérédité** : Supposons  $P(k)$  vraie et  $k + 1 \leq n$  (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  $x_0 \leq \dots \leq x_k$ . 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  $L_k[j] \leq x$ . Quand on s'arrête, on a donc dans  $L[0 : k + 1]$  la valeur  $[x_0, \dots, x_j, x_j, \dots, x_k]$ , puis la ligne 9 assure  $L_{k+1}[0 : k + 1] = [x_0, \dots, x_{j-1}, x, x_j, \dots, x_k]$  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.

— **Conclusion** : Ainsi  $P(k)$  est bien un invariant de boucle.

À 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.

**Q 3.** *Complexité de tri(L)*

**Rédaction 1 :** On note  $C_{tri}(n)$  la complexité de la fonction **tri**( $L$ ) en la longueur  $n$  de la liste  $L$  passée en argument, et  $nw$  le nombre de fois que la boucle **while** est interprétée.

$$C_{tri}(n) = 1 \text{ affectation} + C_{len}(n) + \sum_{i=1}^{i=n-1} [2 \text{ affectations} + nw \times (2 \text{ tests} + 2 \text{ affectations} + 3 \text{ soustractions}) + 1 \text{ affectation}]$$

On peut prendre l'hypothèse ici que toutes les opérations élémentaires ont la même valeur en terme de complexité, et que  $C_{len}(n) = O(n)$ .

Dans le pire des cas,  $nw = i$ , la liste est triée dans l'ordre décroissant.

Alors  $C_{tri}(n) = 1 + O(n) + \sum_{i=1}^{i=n-1} [2 + i \times (2 + 2 + 3) + 1] = 1 + O(n) + 3 \times \sum_{i=1}^{i=n-1} 1 + 7 \times \sum_{i=1}^{i=n-1} i$ .

Donc, dans le pire des cas,  $C_{tri}(n) = 1 + O(n) + 3 \times (n - 1) + 7 \cdot \frac{(n-1) \cdot (n-2)}{2} = O(n^2)$ . C'est une *complexité quadratique*.

Dans le meilleur des cas,  $nw = 1$ , la liste est triée dans l'ordre croissant.

Alors  $C_{tri}(n) = 1 + O(n) + \sum_{i=1}^{i=n-1} [2 + 1 \times (2 + 2 + 3) + 1] = 1 + O(n) + 10 \times \sum_{i=1}^{i=n-1} 1$ .

Donc, dans le meilleur des cas,  $C_{tri}(n) = 1 + O(n) + 10 \times (n - 1) = O(n)$ . C'est une *complexité linéaire*.

**Rédaction 2 :** 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  $2 + \sum_{i=1}^{n-1} (1 + 2 + 4 + 0 + 1) = 8 \cdot n - 5 = O(n)$  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.  $8 \cdot n - 5 + \frac{5 \cdot (n-1) \cdot n}{2} = O(n^2)$  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  $O(n \cdot \ln(n))$ .

#### Q 4. Fonction `tri_chaine()`

```
def tri_chaine(tab):
    '''pré-condition : la liste tab est constitué de couples (chaîne, entier).
    post-condition : la liste tab est triée par ordre croissant de l'entier
    associé à la chaîne.'''
    n= len (tab)
    for i in range (1,n):
        # "tab[:, i] est trié" est invariant de boucle
        j=1
        x=tab[i]
        while 0<j and x[1]<L[j-1][1]:
            # j est variant de boucle
            L[j]=L[j-1]
            j=j-1
        L[j]=x
```

Pour cette première fonction demandée dans le sujet, je prends la peine de correctement la documenter. Ce sera la seule fois dans un sujet bien sûr.

#### Q 5. Clés primaires

Une clé primaire est un ensemble minimal d'attributs dont la valeur définit de manière unique tout enregistrement.

Ainsi aucun 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`, `iso`) est une clé primaire, ainsi que le couple (`annee`, `nom`).

#### Q 6. Première requête

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

### Q 7. Deuxième requête

Attention, il faut deux conditions de jointure.

pays est conservé 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;
```

### Q 8. Troisième requête

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));
```

Une autre solution élégante utilisant les mots clés LIMIT et OFFSET qui ne font pas partie des mots clés cités dans les programmes.

```
SELECT nom FROM palu WHERE annee = 2010
ORDER BY cas DESC LIMIT 1 OFFSET 1;
```

— LIMIT 1 limite le nombre de lignes renvoyées;

— OFFSET 1 indique le nombre de lignes à passer avant de renvoyer les lignes restantes.

### Q 9. Instruction Python

D'après la requête utilisée, `deces201` contient la liste des couples (nom du pays, nombre de décès en 2010). Ainsi, on peut trier par ordre croissant du nombre de décès avec l'instruction suivante.

```
tri_chaine(deces2010)
```

## Deuxième partie

# Modèle à compartiments

### Q 10. Modèle vectoriel

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

### Q 11. Code complété

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

### Q 12. Simulations

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'œil (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.

### Q 13. Code complété

En ligne 7 :

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

En ligne 28 :

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

#### Q 14. *Modification du programme*

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  $\int_0^\pi I(t - s) \cdot h(s) \cdot ds$ . 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 les instructions suivantes.

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

### Troisième partie

## Modélisation dans des grilles

#### Q 15. *Fonction grille(n)*

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.

#### Q 16. *Fonction init(n)*

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

#### Q 17. *Fonction compte(G)*

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  $i$ , dans l'ordre  $(S, I, R, D)$ ).

```
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
```

#### Q 18. *Fonction est\_exposee()*

La fonction `est_exposee()` renvoie un booléen.

#### Q 19. *Code complété*

On rappelle qu'un produit de  $G[i][j] - 1$  est nul si et seulement l'un des termes  $G[i][j]$  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 qui est sur le bord mais pas dans un coin, donc on peut mettre en ligne 12 les instructions suivantes.

```
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 les instructions suivantes.

```
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
```

**Q 20.** *Fonction* suivant(G, p1, p2)

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
```

**Q 21.** *Fonction* simulation(n, p1, p2)

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]
```

**Q 22.** *Exploitation d'une simulation*

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 uniquement des sains et des infectés, toute "case" ayant été à un moment ou à 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  $x_{\text{atteinte}} = (x_2 + x_3)/n^2 = 1 - x_0/n^2$ .

**Q 23.** *Fonction* seuil(Lp2, Lxa)

```
def seuil(Lp2 ,Lxa) :
    mini = 0
    maxi = len (Lp2) - 1
    while maxi - mini > 1 :
        milieu = (mini + maxi) // 2
        if Lxa[milieu] < 0.5 :
            mini = milieu
        else :
            maxi = milieu
    return [Lp2[mini], Lp2[maxi]]
```

À chaque itération :  $\text{milieu} = \lfloor (\text{mini} + \text{maxi})/2 \rfloor$ , avec  $\text{maxi} - \text{mini} > 1$ .

**Option :** Justification de la terminaison de la fonction.

L'écart minimum possible vaut  $\text{maxi} - \text{mini} = 2$ , c'est à dire que  $\text{mini} = \text{maxi} - 2$  au plus haut.

Alors  $\text{milieu} \leq \lfloor (\text{maxi} - 2 + \text{maxi})/2 \rfloor$ ,  $\text{milieu} \leq \text{maxi} - 1$ , c'est à dire  $\text{milieu} < \text{maxi}$ .

Or  $\text{milieu} \geq \text{mini}$ , donc la suite des  $\text{maxi} - \text{mini}$  est strictement décroissante et l'algorithme se termine donc bien.

**Q 24.** *Test de la ligne 8*

On ne peut pas supprimer le test de la ligne 8 pour deux raisons :

- il ne faudrait pas vacciner la "case" initialement infectée (sinon il n'y aura plus de "départ" pour la maladie);
- on veut vacciner exactement une proportion  $q$  de la population, donc il ne faut pas vacciner deux fois la même case et la compter pour deux "cases vaccinées".

**Q 25.** *Appel de `init_vac(5, 0.2)`*

L'appel `init_vac(5, 0.2)` renvoie une grille  $5 \times 5$  i.e. à 25 cases avec exactement une case qui vaut 1,  $nvac = \lfloor 0.2 \times 25 \rfloor = \lfloor 5 \rfloor = 5$  cases qui valent 2 et toutes les autres à 0.