Modélisation de la propagation d'une épidémie

Partie I

Tri et bases de données

Q1. L'application de l'algorithme décrit donne le résultat suivant

```
[5, 2, 3, 1, 4] # Départ
[2, 5, 3, 1, 4]
[2, 3, 5, 1, 4]
[1, 2, 3, 5, 4]
[1, 2, 3, 4, 5]
```

En effet, on prend les éléments un à un en partant de la position 1 et on les « fait couler » jusqu'à ce qu'ils touchent un élément plus petit, faisant par la même occasion « remonter » les éléments les plus grands vers la fin de la liste.

Q2. Prenons comme indiqué par l'énoncé la propriété suivante comme invariant de boucle $\mathscr{P}$(`i`):
« la liste `L[0:i+1]` (avec la convention Python) est triée par ordre croissant à l'issue de l'itération `i` »

Initialisation: Regardons ce qui se passe pour `i` $= 1$. La variable `x` prend la valeur de `L[1]` et la boucle `while` ne s'exécute qu'une seule fois si `L[1]` $<$ `L[0]` en procédant à l'échange et zéro fois sinon. Ainsi, si la liste `L[0:2]` avait été à l'envers, elle est remise en place et sinon on ne fait rien, elle est déjà triée. On a bien que `L[0:i+1]` $=$ `L[0:2]` est triée par ordre croissant en sortie de boucle.

Hérédité: On suppose la propriété vraie au rang `i` et on veut la démontrer au rang `i+1`. Avant l'entrée dans la boucle `while`, `x` prend la valeur `L[i+1]`. Puis, on fait des échanges vers la gauche tant que `x` est plus petit que l'élément courant en décalant l'élément courant une case à droite. On l'insère donc dans la liste `L[0:i+1]` à sa place. Comme par hypothèse `L[0:i+1]` était triée, on a bien `L[0:i+2]` triée à la fin du tour de boucle

Comme la terminaison est assurée par le fait que la boucle `for` de la ligne 3 est bornée et que la boucle `while` de la ligne 6 est nécessairement tôt ou tard arrêtée par la décrémentation de `j` et la vérification `j>0`, on sait qu'à la fin de la dernière boucle (soit `i` $=$ `n-1`), la liste `L[0:n]` $=$ `L` est triée, ce qui démontre la correction de l'algorithme.

Q3. Dans le meilleur des cas, la liste est déjà triée de sorte que la boucle `while` ne s'exécute jamais. La complexité est donc linéaire en nombre n d'éléments dans la liste: en $\mathrm{O}(n)$.
Dans le pire des cas, la liste est triée de manière décroissante de sorte que la boucle `while` s'exécute `i` fois au `i`$^{\text{e}}$ tour de boucle. On fait donc de l'ordre de $\sum_{i=1}^{n-1} i = \mathrm{O}(n^2)$ opérations. La complexité est quadratique dans ce cas.

Q4. Une manière de faire est la suivante

```
1 def tri_chaine(L):
2     n = len(L)
3     for i in range(1,n):
4         j = i
5         x = L[i]
6         while 0 < j and x[1] < L[j-1][1]: # comparaison sur les entiers
7             L[j] = L[j-1]
8             j = j-1
9         L[j] = x
```

Q5. Aucun de ces attributs ne peut servir de clé primaire puisque plusieurs lignes ont la même valeur (première et deuxième pour `nom` et `iso`, deuxième et troisième pour `annee`). Il est peu probable (mais pas impossible) que les attributs `cas` et `deces` soient dupliqués, mais il n'est pas raisonnable de s'appuyer dessus pour caractériser une entrée. En revanche, logiquement, les couples (`nom`,`annee`) ou (`iso`,`annee`) devraient être une bonne clé primaire puisque chaque pays ne devrait avoir qu'un unique enregistrement par année.

Q6. La requête suivante permet d'obtenir les informations recherchées

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

Q7. Pour cette requête, il faut connaître à la fois le nombre de cas (table `palu`) et la population totale du pays (table `demographie`). Il est donc nécessaire d'opérer une jointure entre ces deux tables (à la fois sur le nom du pays et sur l'année regardée). On va rajouter le nom du pays en plus du taux d'incidence sinon on ne pourra pas savoir à quel pays correspond quelle ligne.

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

Q8. Une manière d'effectuer cette requête nécessite l'utilisation de sous-requêtes. On va d'abord déterminer le maximum de cas de paludisme pour l'année 2010

```
MAXI = (SELECT MAX(cas) FROM palu WHERE annee=2010)
```

Puis on sélectionne les pays qui ont strictement moins de cas

```
SOUS_ENSEMBLE = (SELECT nom,cas FROM palu WHERE annee=2010 AND cas < MAXI)
```

On cherche alors le maximum de ce sous-ensemble

```
MAXI2 = (SELECT MAX(cas) FROM SOUS_ENSEMBLE)
```

Et finalement le pays correspondant

```
SELECT nom FROM SOUS_ENSEMBLE WHERE cas = MAXI2
```

Q9. Il suffit d'appliquer la fonction `tri_chaine` définie précédemment. Comme celle-ci modifie directement la liste donnée en argument, rien ne sert d'affecter le résultat (égal à `None` de toutes façons vu qu'il n'y a pas de `return` dans la fonction initiale).

```
1 tri_chaine(deces2010)
```

Partie II

Modèle à compartiments

Q10. Le vecteur X s'écrit $X = (S, I, R, D)$ et la fonction f est telle que

$$f : \begin{cases} \mathbb{R}^4 & \longrightarrow \mathbb{R}^4 \\ (x_0, x_1, x_2, x_3) & \longmapsto (-r\,x_0\,x_1, r\,x_0\,x_1 - (a+b)\,x_1, a\,x_1, b\,x_1) \end{cases}$$

Q11. La définition de la fonction doit se faire comme suit en Python

```
1  def f(X):
2      global r,a,b
3      return [-r*X[0]*X[1], r*X[0]*X[1] - (a+b)*X[1], a*X[1], b*X[1]]
```

Q12. La simulation pour N = 7 a été faite avec un pas d'intégration très grand ($\Delta t = 3{,}6$ pas de temps), d'où de fortes erreurs qui s'accumulent et la présence d'oscillations induites par ces erreurs (voir les losanges ou les carrés par exemple). Au contraire, pour N = 250, on a $\Delta t = 0{,}1$ pas de temps et les erreurs sont bien moins importantes, d'où un comportement plus « lisse ». Néanmoins, on n'a rien sans rien et la simulation à N = 250 va demander environ 36 fois plus de calculs que celle pour N = 7. On remarque qu'outre les fortes oscillations, la simulation à N = 7 semble indiquer que la population infectée peut parfois devenir négative, ce qui est pour le moins problématique à interpréter...

Partie III

Modélisation dans des grilles

Q13. La fonction `grille` renvoie une liste contenant $n+2$ listes de chacune $n+2$ éléments, les cases sur les bords ayant été remplies par des -1 alors que les cases centrales ont été remplies par des 0. Pour le cas $n = 3$, on a une grille de 5 par 5 avec les 9 cases centrales mises à 0. De manière générale, si le paramètre vaut n, alors la population saine initiale est constituée de n^2 individus.

−1	−1	−1	−1	−1
−1	0	0	0	−1
−1	0	0	0	−1
−1	0	0	0	−1
−1	−1	−1	−1	−1

Q14. La fonction demandée peut s'écrire sous la forme

```
1  def init(n):
2      G = grille(n)          # On prend une population vierge
3      i = rd.randrange(n)    # Tirage de l'abscisse
4      j = rd.randrange(n)    # et de l'ordonnée
5      G[i+1][j+1] = 1        # Innoculation du virus (hors bords)
6      return G               # Renvoi de la grille initialisée
```

Q15. En faisant attention à ne pas compter les bords, on a

```
1  def compte(G):
2      C = [0,0,0,0]                    # Initialisation des compteurs
3      for i in range(len(G)):          # Boucle sur les lignes
4          for j in range(len(G[0])):   # et sur les colonnes
5              if G[i][j] >=0:          # On vire les cases du bord
6                  C[G[i][j]] += 1      # et on incrémente le bon compteur
7      return C                         # Renvoi des différents compteurs
```

Q16. La fonction renvoie soit le booléen `False` (ligne 2), soit le résultat de l'évaluation de l'expression `(prod == 0)` qui renvoie elle aussi un booléen (`True` si la grandeur calculée est effectivement nulle, `False` sinon).

Q17. La fonction commence par regarder si on se trouve sur un bord, auquel cas elle renvoie `False` puisqu'un bord ne peut pas être exposé à la maladie. Si on n'est pas sur un bord, on regarde les cases dans les lignes voisines (`i-1`, `i` et `i+1`) et les colonnes voisines (`j-1`, `j` et `j+1`) à l'exception de la case courante (qui vérifie à la fois `k==i` et `l==j`). On regarde donc bien le produit de la valeur de la grille moins un (donc qui vaut zéro quand la case en question est infectée) sur toutes les 8 cases du voisinage immédiat de la case courante. Si ce produit est nul, au moins une case du voisinage est infectée.

Q18. Si le programmeur utilise correctement la fonction (c'est-à-dire vérifie en amont qu'il n'est pas sur une case de bord ou se débrouille pour donner pour i et j des valeurs non sur les bords), la ligne n'est pas nécessaire.

Q19.

```
import copy
def suivant(G,p1,p2):
    new_G = copy.deepcopy(G)          # Copie de la matrice
    for i in range(len(G)):           # Boucle sur les lignes
        for j in range(len(G[0])):    # et les colonnes
            # Si on est malade et qu'on n'a pas de chance
            if G[i][j] == 1 and bernoulli(p1)==1:
                new_G[i][j] = 3       # on meurt.
            else:                     # Et si on a de la chance
                new_G[i][j] = 2       # on survit
            # Si on est sain, en contact avec un ou des malades,
            # et qu'on n'a pas de chance,
            if G[i][j] == 0 and est_exposee(G,i,j) and bernoulli(p2)==1:
                new_G[i][j] = 1       # on se fait infecter.
    return new_G
```

À noter que la copie de la matrice est absolument nécessaire car sinon il peut arriver que l'on tue un malade avant qu'il ne puisse avoir une chance de contaminer ses voisins de droite et d'en-dessous.

Q20.

```
def simulation(n,p1,p2):
    G = init(n)                 # Initialisation (population de n**2)
    while compte(G)[1] != 0:    # Tant qu'il y a des malades,
        G = suivant(G,p1,p2)    # on regarde l'état suivant.
    return [ni/(n**2) for ni in compte(G)] # On renvoie les fractions
```

Q21. En fin de simulation, les infectés sont soit morts, soit guéris donc $x_1 = 0$. Les quatre états étant les seuls accessibles, on doit avoir $x_0 + x_1 + x_2 + x_3 = 1$, soit

```
x_atteinte = 1-x0 # ou = x1+x2+x3 (avec x1 normalement nul en fin de simulation)
```

Q22. Il s'agit d'une variante de la dichotomie classique vue en cours. Par exemple,

```
def seuil(Lp2,Lxa):
    gauche,droite = 0,len(Lp2)-1    # On se place au deux extrémités
    while droite - gauche > 1:      # Tant qu'on n'est pas côte à côte,
        milieu = (droite+gauche)//2 # on regarde au milieu
        if Lxa[gauche] <= 0.5 <= Lxa[milieu]: # Si c'est à gauche,
            droite = milieu         # on décale la borne de droite
        else:                       # sinon c'est à droite
            gauche = milieu         # et on décale la borne de gauche.
    return Lp2[gauche],Lp2[droite]  # On renvoie les valeurs d'encadrement.
```

Appliquée sur les listes menant au graphique de l'énoncé, on obtient

```
>>> seuil(Lp2,Lxa)
(0.30612244897959179, 0.32653061224489793)
```