# Problème – *Modélisation de la propagation d'une épidémie*

## Partie I – Tri et bases de données

**1.** On obtient successivement $n = 5$ puis :

- pour $i = 1$, $x = 2$ et $L$ devient $[2, 5, 3, 1, 4]$
- pour $i = 2$, $x = 3$ et $L$ devient $[2, 3, 5, 1, 4]$
- pour $i = 3$, $x = 1$ et $L$ devient $[1, 2, 3, 5, 4]$
- pour $i = 4$, $x = 4$ et $L$ devient $[1, 2, 3, 4, 5]$

**2.** Notons $L_0$ la liste initiale, $n$ sa longueur et, pour $i \in [\![1, n-1]\!]$

$P(i)$ : "à l'issue de l'itération pour la valeur $i$, la valeur $L_i$ de la variable $L$ est telle que $L_i[0 : i+1]$ est la liste initiale $L_0[0 : i+1]$ triée dans l'ordre croissant, et $L_i[i+1 :] = L_0[i+1 :]$".

Pour $i = 1$, à l'entrée dans la boucle `for`, $j$ prend la valeur 1, $x$ la valeur $L[1]$ et l'on passe au plus une fois dans la boucle `while` :

- si $x \geq L[0]$, on n'entre pas dans la boucle et $L[0 : 2]$ reste inchangée puisqu'à la ligne 9 on remet $x$ à sa place initiale, de sorte que dans ce cas $L_1[0 : 2] = L_0[0 : 2]$ est bien triée ;
- si $x < L[0]$, on entre dans la boucle, $L[1]$ prend la valeur $L[0]$, $j$ est décrémenté, on sort de la boucle `while` et $L[0]$ prend la valeur $x$, de sorte que $L_1[0 : 2] = [L_0[1], L_0[0]]$ est bien la liste $L_0[0 : 2]$ triée.

Comme par ailleurs $L[2 :]$ n'a pas été modifiée, $P(1)$ est vrai.

Je suppose alors $i \in [\![2, n-1]\!]$ tel que $P(i-1)$ soit vrai. Nous repassons alors dans la boucle `for` pour la valeur $i$.

Comme $j$ prend initialement la valeur $i$ et ne peut que décroître, seules les valeurs de $L[: i+1]$ pourront être modifiées durant ce passage, j'aurai donc bien $L_i[i+1 :] = L_{i-1}[i+1 :] = L_0[i+1 :]$ (d'après $P(i-1)$).

De plus, toujours d'après $P(i-1)$, j'ai aussi $L[: i] = [x_0, \ldots, x_{i-1}]$ où $x_0, \ldots, x_{i-1}$ sont les valeurs de $L_0[: i]$ triées par ordre croissant. Au début de l'itération $j$ prend la valeur $i$ et $x$ la valeur $L[i] = L_0[i]$ (d'après $P(i-1)$).

Arrive alors la boucle `while`, dont on sort après $i$ itérations au maximum (car $j$ est décrémenté à chaque passage et l'on sort au pire pour $j = 0$). Ladite boucle `while` décale vers la droite les éléments plus grands que $x$ et l'on en sort dès que $j = 0$ ou $L[j-1] \leq x$. On a alors dans $L[: i+1]$ la liste $[x_0, \ldots, x_{j-1}, x_j, x_j, \ldots, x_{i-1}]$ (où $x_j$ se trouve aux indices $j$ et $j+1$, suite au décalage), puis la ligne 9 installe $x$ à la bonne place (en début de liste si l'on est sorti avec $j = 0$, juste après $x_{j-1}$ sinon, alors que $x_{j-1} \leq x < x_j$ par définition de la valeur de $j$ à la sortie de la boucle `while`).

Comme $x$ est justement la valeur $L_0[i]$, j'ai bien dans $L_i[: i+1]$ les valeurs de $L_0[: i+1]$ triées par ordre croissant, donc $P(i)$ est vrai. Je viens de montrer par récurrence que

$P(i)$ est un invariant de boucle.

Le fait que $P(n-1)$ soit vrai permet de conclure (il n'y a plus rien dans $L_{n-1}[n :]$ !).

`tri(L)` trie la liste `L`.

**3.** Dans le meilleur cas, la liste est déjà triée par ordre croissant et le contenu de la boucle `while` n'est jamais exécuté : on obtient alors une complexité en $O(n)$.

Dans le pire des cas, la liste est triée dans l'ordre inverse, pour tout $i$ on passe $i$ fois dans la boucle `while`, donc on obtient une complexité en $O\left(\sum_{i=0}^{n-1} i\right)$, soit en $O\left(n^2\right)$.

L'algorithme de tri fusion est plus rapide dans le pire des cas car il s'exécute **dans tous les cas** avec une complexité en $O(n \ln n)$.

**4.** Pour adapter le tri par insertion proposé dans l'énoncé, `tri`, il suffit de comparer les secondes valeurs des listes présentes dans `L` :

```
def tri(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
```

**5.** Rappelons qu'une clé primaire est une famille minimale d'attributs dont la valeur définit de manière unique tout enregistrement. Ici aucun attribut seul n'est une clé primaire pour la table `palu` : un même pays (`nom` ou `iso`) peut apparaître dans plusieurs enregistrements (avec des années différentes), une même année peut apparaître plusieurs fois pour des pays différents et seul le hasard pourrait faire qu'un nombre de cas ou de décès puisse rester unique !

En revanche, les couples (`annee`, `nom`) ou (`annee`, `iso`) peuvent servir de clé primaire.

**6.** Requête standard ; on peut écrire :

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

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

```
SELECT pays, 100000.0*cas/pop AS taux_incidence_2011
FROM palu JOIN demographie ON iso = pays AND annee = periode
WHERE annee = 2011;
```

**8.** Pour l'année 2010, je cherche le plus grand nombre de nouveaux cas de paludisme, je crée une nouvelle table virtuelle sans ce dernier où je cherche le pays ayant eu le maximum de cas (parmi ceux qui restent)...

```
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));
```

Autre version, plus concise mais utilisant `LIMIT` et `OFFSET` qui sont *a priori* hors programme...

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

Enfin une troisième version, astucieuse et sans `OFFSET` :

```
SELECT nom FROM (SELECT nom, cas FROM palu WHERE annee = 2010
                 ORDER BY cas DESC LIMIT 2)
           ORDER BY cas LIMIT 1
```

**9.** D'après la requête utilisée, `deces2010` contient la liste des couples (nom du pays, nombre de décès en 2010). Alors, grâce au programme du **4.**, on peut trier par ordre croissant du nombre de décès par

```
tri_chaine(deces2010).
```

*Remarquons qu'avant la conversion vers Python, on aurait pu faire directement trier le résultat via la requête SQL...*

## Partie II – Modèle à compartiments

**10.** On peut prendre

$$\boxed{X = (S, I, R, D) \text{ et } f : (S, I, R, D) \in \mathbb{R}^4 \mapsto \big(-rSI, rSI - (a+b)\,I, aI, bI\big).}$$

**11.** Il suffit de remplacer 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]])
```

**12.** 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. La simulation avec $N = 250$ fournit beaucoup plus de points d'où un tracé qui semble continu à l'œil nu.

**13.** On remplace la ligne 5 (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 l'on place en ligne 26

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

**14.** Il suffit de remplacer la ligne `Itau = XX[i - p][1]` du code précédent par :

```
Itau = dt * sum([XX[i-j][1] * h(j*dt) for j in range(p)])
```

*(Je me suis permis d'utiliser la fonction* `sum` *de Python qui renvoie la somme des éléments d'une liste, on pourrait bien sûr la remplacer par une boucle banale...)*

## Partie III – Modélisation dans des grilles

**15.** Chaque passage dans la boucle `for` externe construit une liste contenant $n$ fois la valeur 0 et l'ajoute à la liste $M$, donc

`grille(n)` renvoie une liste de $n$ listes de $n$ éléments valant tous 0.

**16.** J'initialise $G$ à une grille pleine de 0, je choisis au hasard un couple de coordonnées et je mets à 1 1a case correspondante.

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

**17.** Je parcours toute la grille et j'incrémente pour chaque valeur trouvée son nombre d'occurrences, stocké dans la liste `nombres`, `nombres[k]` contenant naturellement le nombre de cases dans l'état `k` :

```
def compte(G):
    nombre = [0, 0, 0, 0]
    for L in G:
        for k in L:
            nombre[k] += 1
    return nombre
```

**18.** Je parcours les couples `[i,j]` présents dans la liste `L` et je renvoie `True` dès que je trouve un 1 dans la case `G[i][j]`. Je n'oublie pas de renvoyer `False` si je n'ai trouvé aucun 1 !

```
def zone_infectee(G, L):
    for [i,j] in L:
        if G[i][j] == 1:
            return True
    return False
```

**19.** Il s'agit selon les différentes positions possibles de la case examinée, par rapport aux bords et aux coins de la grille, de tester si l'une des cases voisines est infectée. J'utilise donc la fonction précédente.

- En ligne 12, c'est le cas d'une case de la première ligne, mais pas l'un des coins (traités en ligne 4 et en ligne 6), ladite case a donc 5 voisines :

```
return zone_infectee(G, [[0,j-1], [1,j-1], [1,j], [1,j+1], [0,j+1]])
```

- En ligne 20, c'est le "cas général" d'une case qui n'est sur aucun bord et a donc 8 voisines :

```
return zone_infectee(G,
[[i-1,j-1], [i,j-1], [i+1,j-1], [i+1,j], [i+1,j+1], [i,j+1], [i-1,j+1], [i-1,j]])
```

**20.** J'ai besoin de la grille à l'instant présent pour déterminer les zones infectées. Je crée donc une nouvelle grille `Gs` et je détermine le statut de chaque case à l'instant suivant en appliquant la "règle du jeu" :

- une case rétablie ou décédée reste dans le même état
- une cases infectée devient décédée avec la probabilité $p_1$, devient rétablie sinon
- une case saine exposée devient infectée avec la probabilité $p_2$, reste saine sinon

```
def suivant(G, p1, p2):
    n = len(G)
    Gs = grille(n)                                  #nouvelle grille
    for i in range(n):
        for j in range(n):
            if G[i][j] == 1:                        #infecté
                Gs[i][j] = 2 + bernoulli(p1)
            elif G[i][j] == 0 and est_exposee(G,i,j): #sain
                Gs[i][j] = bernoulli(p2)
            else:                                   #rétabli ou décédé
                Gs[i][j] = G[i][j]
    return Gs
```

**21.** Comme indiqué dans l'énoncé, j'initialise la grille avec toutes les cases saines, sauf une qui est infectée. Ensuite la grille ne peut évoluer que s'il reste au moins une case infectée. Je pourrais appeler `compte` à chaque étape, mais il sera plus économique d'utiliser une fonction `est_infectee(G)`, écrite suivant le même principe que `zone_infectee` mais balayant toute la grille. Je n'appelle `compte` qu'à la fin pour renvoyer le résultat demandé.

```
def est_infectee(G):
    for L in G:
        if 1 in L:
            return True
    return False

def simulation(n, p1, p2):
    G = init(n)
    while est_infectee(G):
        G = suivant(G, p1, p2)
    return [x/n**2 for x in compte(G)]
```

Une case infectée change toujours de statut à l'étape suivante et ensuite n'évolue plus. Donc le pire des cas est celui où exactement une nouvelle case est infectée à chaque étape et, dans ce cas le nombre d'étapes est $n^2$ ($n^2 - 1$ infections puis une dernière étape pour savoir si la dernière case infectée se rétablit ou décède...).

La simulation se termine en $n^2$ étapes au maximum.

**22.** Nous avons vu que la simulation se termine lorsqu'il n'y a plus aucune case infectéé, donc

`x1 = 0.`

De plus, par définition, la somme des 4 proportions vaut 1 puisque l'on recense tous les états possibles et qu'ils s'excluent mutuellement, donc

`x0 + x1 + x2 + x3 = 1.`

`x0` donne la proportion des cases qui sont restées saines, donc

La proportion des cases qui ont été atteintes est `x2 + x3`, aussi égale à `1 - x0`.

**23.** Pour le plaisir, écrivons une version avec fonction auxiliaire récursive, qui renverra en général un intervalle entre deux valeurs consécutives de la liste `Lp2`, **sauf** dans le cas où l'on tombe pile sur la valeur 0.5, qu'il faut traiter puisque l'énoncé demande "la plus grande précision possible" !

```
def seuil (Lp2, Lxa):
    def dicho (indmin, indmax):
        if indmax - indmin == 1:
            if Lxa[indmin] == 0.5:
                return [Lp2[indmin], Lp2[indmin]]
            else:
                return [Lp2[indmin], Lp2[indmax]]
        else:
            indmil = (indmax + indmin) // 2
            if Lxa[indmil] > 0.5:
                return dicho(indmin, indmil)
            else:
                return dicho(indmil, indmax)

    return dicho(0, len(Lp2) - 1)
```

Les appels récursifs se terminent car la différence `indmax - indmin` diminue strictement tant qu'elle est au moins égale à 2 et l'on s'arrête lorsqu'elle vaut 1. De plus une récurrence immédiate montre que l'on a en permanence

`Lxa[indmin] <= 0.5 < Lxa[indmax]`

d'où le choix de l'intervalle renvoyé.

**24.** 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 aucune propagation).
- 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".

**25.** Dans le cas proposé, $nvac = \lfloor 0.2 * 5^2 \rfloor = \lfloor 5 \rfloor$ et donc

L'appel `init_vac(5, 0.2)` renvoie une grille $5 \times 5$
avec exactement une case à 1, 5 cases à 2 et toutes les autres à 0.

## Exercice bonus – *Le compte est bon*

**Question 1** – J'adopte le principe proposé par l'énoncé...

Je commence par créer la double boucle souhaitée, la boucle interne partant de `i+1` pour prendre une seule fois chaque paire {`i`, `j`} d'indices.
Ensuite (lignes 9 et 10) j'échange `a` et `b` si besoin, pour avoir `a >= b` ; c'est important pour la soustraction et la division...
La ligne 11 crée par concaténation la liste `L2` contenant les valeurs de `L` sauf celles d'indices `i` et `j`.
Ligne 12, je traite l'addition comme indiqué dans l'énoncé.
Lignes 13 et 14, je traite la multiplication, en écartant les cas où l'une des valeurs vaut 1, ce qui n'amènerait aucune nouvelle valeur dans `L`.
Lignes 15 et 16, je traite la soustraction, en écartant le cas où les deux valeurs sont égales, car on ne veut pas de zéro.

Enfin, lignes 17 et 18, je traite la division de `a` par `b`, à effectuer uniquement lorsque `b` est au moins égal à 2 et qu'elle "tombe juste", puisque l'on veut rester dans $\mathbb{N}$.

```
1  def compte(L0, N):
2      def aux(L, calc):
3          if N in L:
4              solutions.append(calc)
5          else:
6              for i in range(len(L)-1):
7                  for j in range(i+1, len(L)):
8                      a, b = L[i], L[j]
9                      if a < b:
10                         a, b = b, a
11                     L2 = L[:i] + L[i+1:j] + L[j+1:]
12                     aux(L2+[a+b], calc+' '+str(a)+'+'+str(b)+'='+str(a+b))
13                     if a != 1 and b != 1:
14                         aux(L2+[a*b], calc+' '+str(a)+'*'+str(b)+'='+str(a*b))
15                     if a != b:
16                         aux(L2+[a-b], calc+' '+str(a)+'-'+str(b)+'='+str(a-b))
17                     if b != 1 and a%b == 0:
18                         aux(L2+[a//b], calc+' '+str(a)+'/'+str(b)+'='+str(a//b))
19
20     solutions = []
21     aux(L0, '')
22     return solutions
```

Les appels récursifs se terminent bien, puisque la longueur de `L` diminue strictement à chaque appel récursif et qu'il y en a plus aucun quand la liste est de longueur 0 ou 1. Cela dit, on n'a pas mieux pour la complexité qu'un majorant en $O\left(4^n\right)$ où $n$ est la longueur de la liste initiale `L0`.

Enfin, chaque chaîne de calcul ajoutée dans la liste `solutions` donne bien une solution, puisque l'ajout se fait au moment où l'objectif `N` apparaît dans la liste des résultats "intermédiaires" disponibles, alors même que l'on vient d'ajouter à la fin de `calc` le dernier calcul dont le résultat est `N`. Et toutes les solutions sont explorées puisque l'on envisage toutes les opérations envisageables avec les valeurs disponibles dans `L`, en tout cas celles qui donneront une nouvelle valeur élément de $\mathbb{N}^*$.

**Question 2** – Pour le cas où $N$ est hors d'atteinte, il suffit d'implémenter une boucle qui lance la fonction précédente pour des valeurs de $N$ qui s'éloignent pas à pas de $N$, tant que la liste de solutions est vide...

```
1  def compte_proche(L0, N):
2      solutions = compte(L0, N)
3      if solutions != []:
4          return "Compte exact", solutions
5      else:
6          ecart = 1
7          while True:
8              solutions = compte(L0, N + ecart)
9              if solutions != []:
10                 return "Au plus proche " + str(N + ecart), solutions
11             solutions = compte(L0, N - ecart)
12             if solutions != []:
13                 return "Au plus proche " + str(N - ecart), solutions
14             ecart += 1
```

La boucle `while True` est bien sûr à utiliser avec précaution ! Mais ici elle sera nécessairement interrompue par un `return`, puisque l'on finira bien par trouver un entier accessible, ne serait-ce que l'une des valeurs de `L0` !