# CORRIGÉ DM INFO n°2 (MINES IPT 2016)

## Partie I. Tri et bases de données

❏ **Q1.** Initialement, `L=[5, 2, 3, 1, 4]`. À la fin de chaque boucle on a les résultats suivants :

| `i` | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| `L` | `[2, 5, 3, 1, 4]` | `[2, 3, 5, 1, 4]` | `[1, 2, 3, 5, 4]` | `[1, 2, 3, 4, 5]` |

❏ **Q2.** On démontre la propriété demandée par récurrence sur `i`.

– Il faut comprendre que le cas `i=0` correspond à la situation avant la 1ère itération.

La propriété `L[0:i+1]` est triée signifie alors que la liste formée du seul élément `L[0]` est triée, c'est évidemment vérifié.

*Je rappelle que l'extraction d'une sous-liste par l'instruction `L[a:b]` donne la liste de 1er terme `L[a]` et comportant $b-a$ éléments, c'est-à-dire se terminant à `L[b-1]`.*

– Suppososn alors la propriété vérifiée au rang `i` (avec $i < n-1$). Alors, au début de la $i+1$-ème itération, la liste `L[0:i+1]` est triée. Notons

$$\texttt{L[0:i+1]} = [x_0, x_1, \dots, x_i] \text{ avec } x_k \leqslant x_{k+1} \ .$$

On fait alors `j = i+1` et `x=L[i+1]`.

– si $x \geqslant L[i]$, la boucle `while` n'est pas exécutée, donc la sous-liste `L[0:i+2]` est triée, puisque $L[i+1] \geqslant L[i] \geqslant \cdots$
La propriété est donc vraie à l'ordre `i+1` à la fin de la $i+1$-ème itération.

– sinon, $x < L[i]$ ; la boucle `while` s'exécute, et à la fin de cette boucle, on a

$$\texttt{j=0} \quad \text{ou} \quad \texttt{x} \geqslant \texttt{L[j-1]}$$

et on a décalé les éléments :

$$x_i \longrightarrow L[i+1], \dots, x_j \longrightarrow L[j+1]$$

`L[0:i+2]` est donc devenue

$$\texttt{L[0:i+2]} = [x_0, x_1, \dots, \underset{\uparrow\, j}{x_j}, \underset{\uparrow\, j+1}{x_j}, \dots, \underset{\uparrow\, i+1}{x_i}]$$

puis on fait : `L[j]=x` avec $x \geqslant x_{j-1}$ (dans le cas où $j \neq 0$) et $x < x_j$. On obtient donc

$$\texttt{L[0:i+2]} = [x_0, x_1, \dots, x_{j-1}, x, x_j, \dots, x_i]$$

qui est une liste triée, ce qui démontre la propriété voulue.

*Conclusion :* à la fin de la procédure (`i=n-1`), la propriété est donc encore vraie, cad que `L = L[0:n]` est triée.

❏ **Q3.** Complexité :

– Dans le ;meilleur des cas, la liste est déjà triée ; on ne rentre donc jamais dans la boucle `while`, la procédure se contente de parcourir la liste en faisant des opérations en temps constant à chaque fois (2 affectations et deux tests), donc la complexité est un $O(n)$.

– Dans le pire des cas, la liste initiale est triée en ordre décroissant. Dans ce cas, à chaque étape, la boucle `while` sera exécutées $i$ fois, puisqu'il faut remettre l'élément d'indice $i$ au début.

On aura donc fait, à une constante multiplicative près, $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ opérations, la complexité est en $O(n^2)$.

– On préférera donc par exemple le `tri fusion`, de complexité $O(n \ln n)$.

❏ **Q4.** Pour `tri_chaine`, il suffit de remplacer la ligne 6 par :

```
while 0 < j and x[1] < L[j-1][1].
```

❏ **Q5.** Une `clé primaire` est un attribut, ou un ensemble d'attributs, qui détermine de manière unique un enregistrement.

Ici, aucun attribut seul ne peut être une clé primaire, pusique, par exemple, chaque pays apparaît plusieurs fois avec le même nom.
Cependant, les couples `(nom, année)` ou `(iso, année)` peuvent servir de clé primaire.

❏ **Q6.**

```
SELECT * FROM palu WHERE annee = 2021 AND deces >= 1000
```

❏ **Q7.**

```
SELECT nom, 100000.0*cas/pop AS Taux FROM palu, demographie
        WHERE annee = 2011 AND periode = 2011 AND iso = pays
```

ou bien :

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

❏ **Q8.** Le plus simple est d'utiliser `LIMIT` (nombre total de résultats affichés) avec `OFFSET` (nombre de cas non affichés dans le résultat initial de la requête), les enregistrements ayant été triés par ordre décroissant.

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

Sinon, on est obligé de rechercher l'enregistrement correspondant au maximum des cas, puis de sélectionner l'enregistrement ayant le max des cas parmi ceux différents du précédent, ce qui se fait à l'aide d'ordres `SELECT` imbriqués.

```
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.** La requête de l'énoncé écrite dans le langage de l'algèbre relationnelle ($\pi$ = projection, $\sigma$ = sélection) signifie simplement :

```
SELECT nom, pays FROM palu WHERE annee = 2010
```

Pour trier la liste obtenue, il suffit donc de faire :

`tri_chaine(deces2010)`

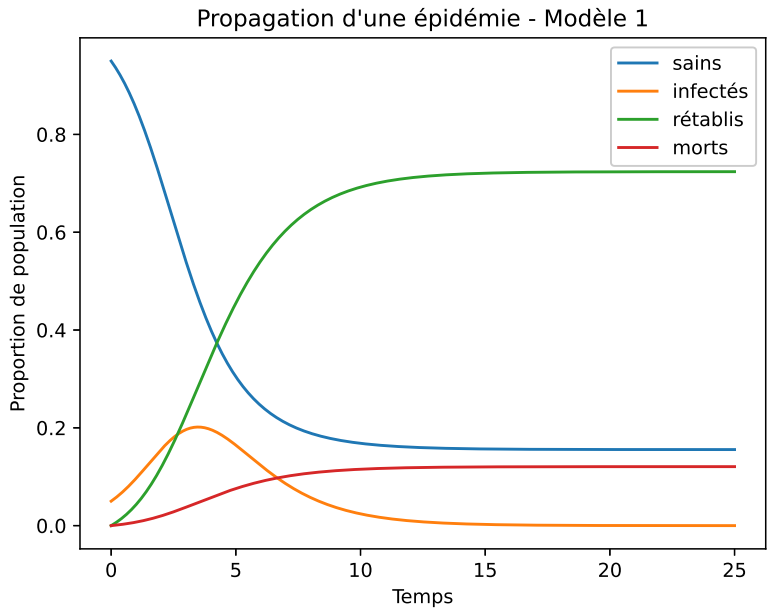
## Partie II. Modèle à compartiments

❏ **Q10.** Voir dans la question suivante.

❏ **Q11.** Ci-dessous le 1er programme de simulation, suivi de la figure obtenue.

```
import numpy as np
import matplotlib.pyplot as plt

# Paramètres virus (variables globales)
r = 1.5 # Plus r est grand et plus la maladie est contagieuse
a = 0.6 # Plus a est grand et plus les malades guérissent
b = 0.1 # Plus b est grand et plus les malades meurent

# Conditions initiales
S0 = 0.95 # proportion initiale d'individus sains
I0 = 0.05 # proportion initiale d'individus infectés
R0 = 0    # proportion initiale d'individus immunisés
D0 = 0    # proportion initiale d'individus morts
X0 = np.array([S0, I0, R0, D0])
```

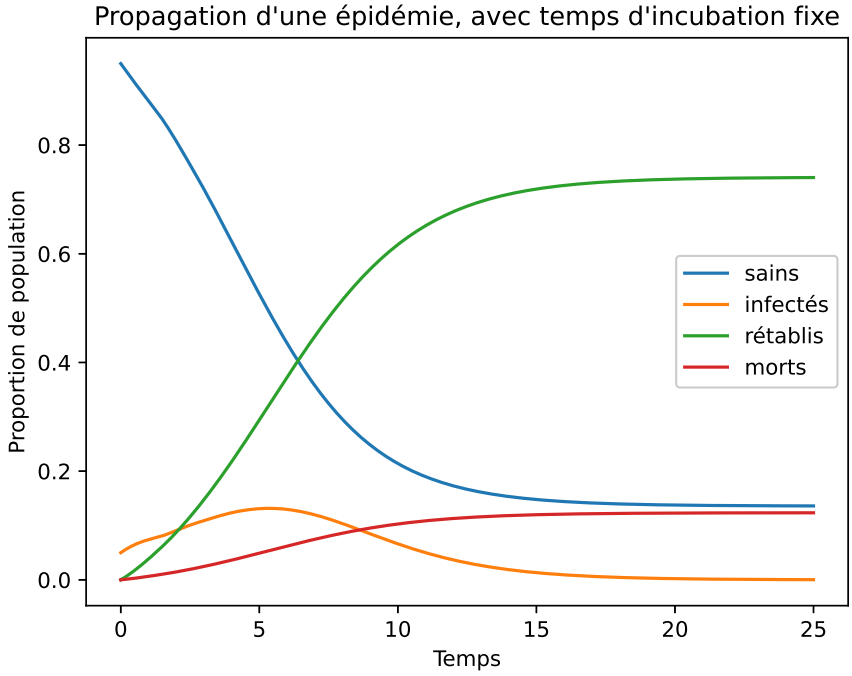
```python
def traceCourbes1(N=100000, tmax=25):

    def f(X):
        """ Fonction définissant le système différentiel """
        S, I, R, D = X       # plus joli que S = X[0] I=X[1] etc...
        return np. array ([-r*S*I, r*S*I -(a+b)*I, a*I, b*I])
        # on renvoie un vecteur numpy et pas une liste  pour pouvoir faire les calculs

    def Euler1(N, tmax):

        dt = tmax/N
        t = 0
        tt =[t]
        X = X0
        XX = [X]
        for i in range(N):
            t += dt
            X = X + dt*f(X) # calcul vectorialisé grâce à Numpy
            # Attention : X += dt*f(X) ne fonctionne pas ici car il y aura une seule
            # adresse pour tous les X(t) donc XX ne contiendra à la fin qu'une seule valeur
            tt.append(t)
            XX.append(X)

        return tt, XX

    tt, XX = Euler1(N, tmax)

    plt.plot(tt, [X[0] for X in XX], label="sains")
    plt.plot(tt, [X[1] for X in XX], label="infectés")
    plt.plot(tt, [X[2] for X in XX], label="rétablis")
    plt.plot(tt, [X[3] for X in XX], label="morts")

    plt.title("Propagation d'une épidémie - Modèle 1")
    plt.xlabel("Temps")
    plt.ylabel("Proportion de population")
    plt.legend()

plt.figure(1)
traceCourbes1()
plt.show()
```



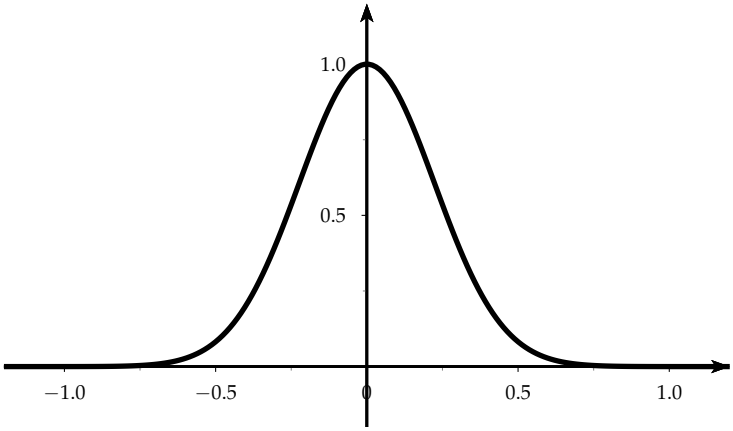
❏ **Q12.** Question bien curieuse ! Il est clair que l'on obtient un tracé plus précis avec plus de points, mais que c'est plus long...

❏ **Q13.** Ci-dessous le second programme de simulation, suivi de la figure obtenue.

```
import numpy as np
import matplotlib.pyplot as plt

# Paramètres virus (variables globales)
r = 1.5 # Plus r est grand et plus la maladie est contagieuse
a = 0.6 # Plus a est grand et plus les malades guérissent
b = 0.1 # Plus b est grand et plus les malades meurent

# Conditions initiales
S0 = 0.95 # proportion initiale d'individus sains
I0 = 0.05 # proportion initiale d'individus infectés
R0 = 0    # proportion initiale d'individus immunisés
D0 = 0    # proportion initiale d'individus morts
X0 = np.array([S0, I0, R0, D0])

def traceCourbes2(N=100000, tmax=25, p=5000):

    def f(X,Itau):
        """ Fonction pour l'équation avec retard.
        Itau est la valeur de I(t-p*dt) càd p pas de temps avant."""
        S, I, R, D = X
        return np.array( [-r*S*Itau, r*S*Itau -(a+b)*I, a*I, b*I])

    def Euler2(N, tmax, p):

        dt = tmax/N
        t = 0
        tt =[t]
        X = X0
        XX = [X]
        for i in range(N):
            t += dt
            if i<p: # t<tau
                Itau = X0[1]
            else:
                Itau = XX[i-p][1]
            # Itau = I(t-tau)
            X = X + dt*f(X, Itau)
            tt.append(t)
            XX.append(X)
        return tt, XX

    tt, XX = Euler2(N, tmax, p)

    plt.plot(tt, [X[0] for X in XX], label="sains")
    plt.plot(tt, [X[1] for X in XX],label="infectés")
    plt.plot(tt, [X[2] for X in XX], label="rétablis")
    plt.plot(tt, [X[3] for X in XX], label="morts")

    plt.title("Propagation d'une épidémie, avec temps d'incubation fixe")
    plt.xlabel("Temps")
    plt.ylabel("Proportion de population")
    plt.legend()

plt.figure(2)
traceCourbes2()
plt.show()
```



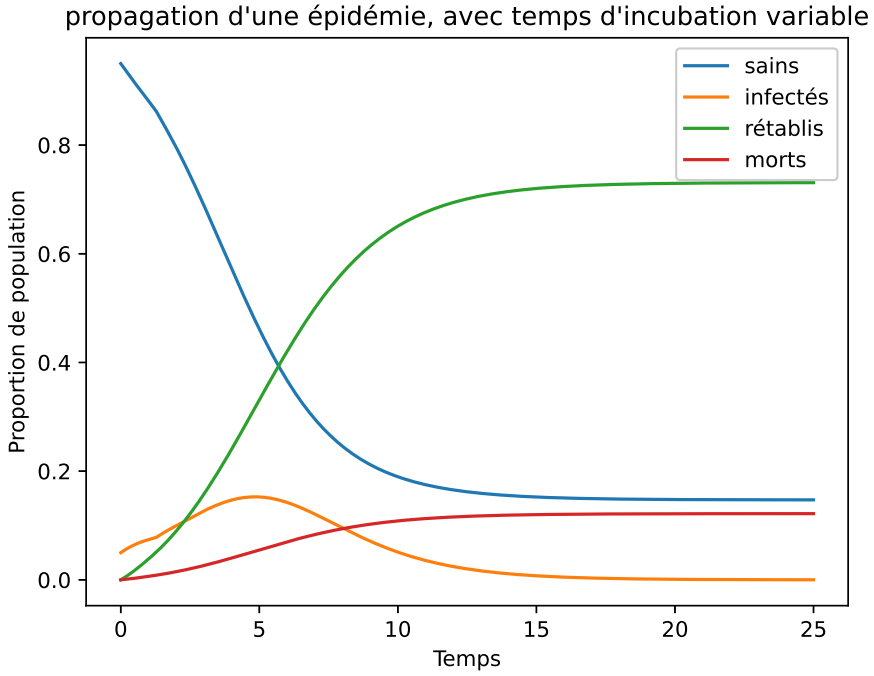

❏ **Q14.** Considérons la fonction $\varphi : x \mapsto \mathrm{e}^{-10x^2}$, que l'on peut considérer comme nulle en dehors de $[-1\,;1]$ et qui ressemble assez à la figure de l'énoncé :




En la translatant et en divisant par son intégrale, on peut obtenir une fonction $h$ comme dans l'énoncé. Cela donne le programme suivant.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import quad

# Paramètres virus (variables globales)
r = 1.5 # Plus r est grand et plus la maladie est contagieuse
a = 0.6 # Plus a est grand et plus les malades guérissent
b = 0.1 # Plus b est grand et plus les malades meurent

# Conditions initiales
S0 = 0.95 # proportion initiale d'individus sains
I0 = 0.05 # proportion initiale d'individus infectés
R0 = 0    # proportion initiale d'individus immunisés
D0 = 0    # proportion initiale d'individus morts
X0 = np.array([S0, I0, R0, D0])

def traceCourbes3(N=100000, tmax=25, p=5000):
```

```
    def f(X,Itau):
        """ Fonction pour l'équation avec retard.
        Itau est la valeur de l'intégrale de l'énoncé"""
        S, I, R, D = X
        return np.array( [-r*S*Itau, r*S*Itau -(a+b)*I, a*I, b*I])

    # Calcul de la fonction h
    dt = tmax/N
    tau = p*dt

    def phi(x):
        # sorte de gaussienne
        if -1<x<1:
            return np.exp(-10*x*x)
        else:
            return 0
    def psi(x):
        # gaussienne translatée
        return phi(x*2/tau-1)

    integrale_de_psi = quad(psi, 0, tau)[0]

    def h(x):
        # la même mais d'intégrale 1
        return psi(x)/integrale_de_psi

    def Euler3(N, tmax, p):

        dt = tmax/N
        t = 0
        tt =[t]
        X = X0
        XX = [X]
        for i in range(N):
            t += dt
            if i<p: # t<tau
                Itau = X0[1]
            else: # calcul de l'intgérale par la méthode des rectangles
                Itau = dt * sum( [XX[i-j][1]*h(j*dt) for j in range(p)])
            X = X + dt*f(X,Itau)
            tt.append(t)
            XX.append(X)
        return tt, XX

    tt, XX = Euler3(N, tmax, p)

    plt.plot(tt, [X[0] for X in XX], label="sains")
    plt.plot(tt, [X[1] for X in XX],label="infectés")
    plt.plot(tt, [X[2] for X in XX], label="rétablis")
    plt.plot(tt, [X[3] for X in XX], label="morts")

    plt.title("propagation d'une épidémie, avec temps d'incubation variable")
    plt.xlabel("Temps")
    plt.ylabel("Proportion de population")
    plt.legend()

plt.figure(3)
traceCourbes3()
plt.show()
```




❏ **Q15.** L'appel à `grille(n)` renvoie une matrice de taille $n \times n$, sous forme d'une liste de $n$ listes de longueurs $n$, et où tous les éléments sont initialisés à $0$.

❏ **Q16.** Voir programme ci-dessous. J'ai également écrit une procédure `init_multiple` qui construit une grille où il y a déjà une proportion $I_0$ de cases infectées ; cela accélère considérablement les calculs pour la suite.

❏ **Q17.** Voir programme ci-dessous.

❏ **Q18.** Voir programme ci-dessous.

❏ **Q19.** La fonction `est_exposee` renvoie un résultat de type booléen (`bool`).

La fonction telle qu"elle est écrite fait beaucoup trop de calculs pour rien ! J'ai donc écrit une procédure `est_exposee_amelioree`, qui ne fait que des tests et qui est plus rapide (car dès que le résultat d'un test est `True`, les autres tests ne sont pas faits à cause du `or`).

❏ **Q20.** Voir programme ci-dessous.

❏ **Q21.** Voir programme ci-dessous. On peut sensiblement augmenter la rapidité du programme en mettant à jour la liste `[n0, n1, n2, n3]` au fur et à mesure, c'est l'objet de la fonction `suivant_ameliore`.

❏ **Q22.** À la fin de la simulation, on a $x_1 = 0$ ; c'est d'ailleurs la condition d'arrêt de la boucle `while` dans la fonction `simulation`.

On a bien sûr : $x_0 + x_1 + x_2 + x_3 = 1$.

La valeur `x_atteinte` est donnée par : $x_{\text{atteinte}} = x_2 + x_3 = 1 - x_0$.

❏ **Q23.** Voir programme ci-dessous.

❏ **Q24.** Voici le programme complet, et les courbes obtenues (pour $n = 1000$, soit $1\,000\,000$ personnes. C'est un peu long...).

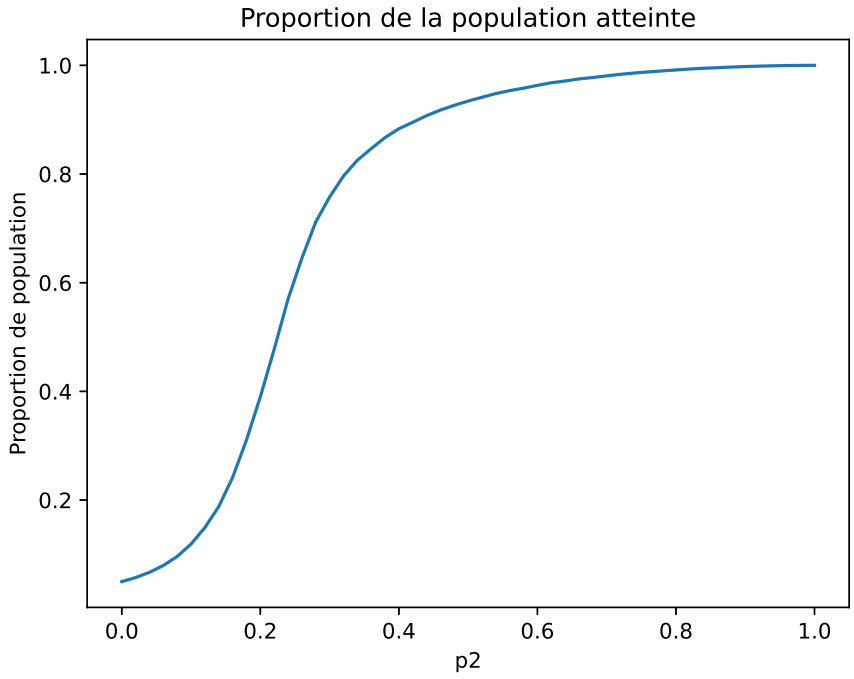
```
import numpy as np
import matplotlib.pyplot as plt
import random as rd
from copy import deepcopy

def grille(n):
    # remplit une grille nxn avec des zéros
    # on pourrait écrire, plus rapidement:
    # return [ [0 for j in range(n)] for i in range(n) ]
    M=[]
```

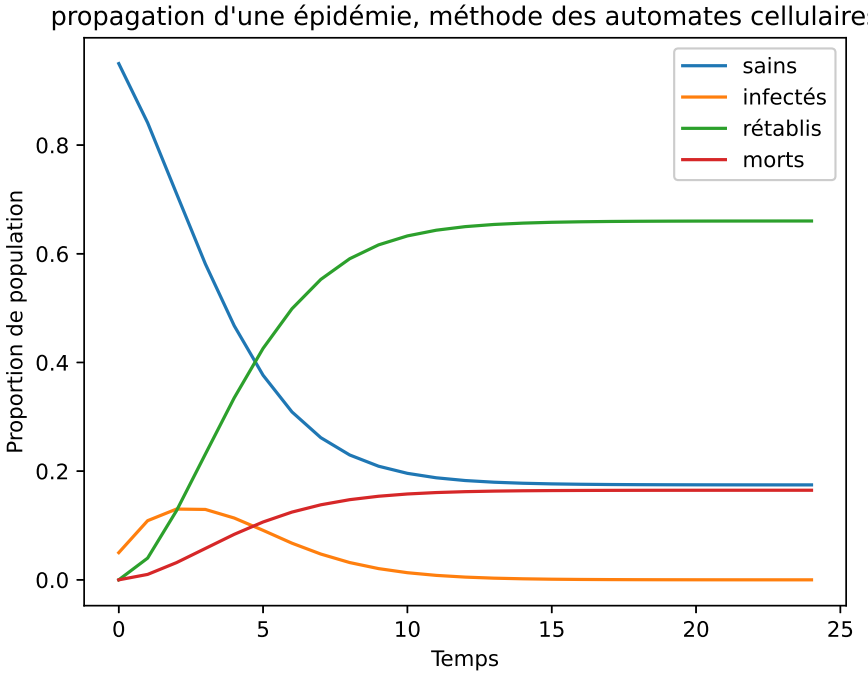
```
    for i in range(n) :
        L=[]
        for j in range(n):
            L.append(0)
        M.append(L)
    return M

# Comme dans l'énoncé, une seule personne infectée:
def init (n):
    G = grille(n)
    i = rd.randrange(n)
    j = rd.randrange(n)
    G[i][j] = 1
    return G

# pour être plus réaliste, on peut supposer que la proportion I0 de la population est infectée
# et cela accélère considérablement les calculs!

def init_multiple(n):
    G = grille(n)
    nb = int(I0*n*n)
    k = 0
    while k < nb:
        i = rd.randrange(n)
        j = rd.randrange(n)
        if G[i][j] == 0:
            G[i][j] = 1
            k += 1
    return G

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

def est_exposee(G, i, j):
    n = len(G)
    if i == 0 and j == 0:
        return (G[0][1]-1)*(G[1][1]-1)*(G[1][0]-1) == 0
    elif i == 0 and j == n-1:
        return (G[0][n-2]-1)*(G[1][n-2]-1)*(G[1][n-1]-1) == 0
    elif i == n-1 and j == 0:
        return (G[n-1][1]-1)*(G[n-2][1]-1)*(G[n-2][0]-1) == 0
    elif i == n-1 and j == n-1:
        return (G[n-1][n-2]-1)*(G[n-2][n-2]-1)*(G[n-2][n-1]-1) == 0
    elif i == 0:
        # complété:
        return (G[0][j-1]-1)*(G[1][j-1]-1)*(G[1][j]-1)*(G[1][j+1]-1)*(G[0][j+1]-1) == 0
    elif i == n-1:
        return (G[n-1][j-1]-1)*(G[n-2][j-1]-1)*(G[n-2][j]-1)*(G[n-2][j+1]-1)*(G[n-1][j+1]-1) =
    elif j == 0:
        return (G[i-1][0]-1)*(G[i-1][1]-1)*(G[i][1]-1)*(G[i+1][1]-1)*(G[i+1][0]-1) == 0
    elif j == n-1:
        return (G[i-1][n-1]-1)*(G[i-1][n-2]-1)*(G[i][n-2]-1)*(G[i+1][n-2]-1)*(G[i+1][n-1]-1) =
    else:
        # complété:
        return (G[i-1][j-1]-1)*(G[i-1][j]-1)*(G[i-1][j+1]-1)*(G[i][j-1]-1)*(G[i][j+1]-1)\
                *(G[i+1][j-1]-1)*(G[i+1][j]-1)*(G[i+1][j+1]-1) == 0
```

```
def est_exposee_amelioree(G, i, j):
    n = len(G)
    if i == 0 and j == 0:
        return G[0][1] == 1 or G[1][1] == 1 or G[1][0] == 1
    elif i == 0 and j == n-1:
        return G[0][n-2] == 1 or G[1][n-2] == 1 or G[1][n-1] == 1
    elif i == n-1 and j == 0:
        return G[n-1][1] == 1 or G[n-2][1] == 1 or G[n-2][0] == 1
    elif i == n-1 and j == n-1:
        return G[n-1][n-2] == 1 or G[n-2][n-2] == 1 or G[n-2][n-1] == 1
    elif i == 0:
        return G[0][j-1] == 1 or G[1][j-1] == 1 or G[1][j] == 1\
               or G[1][j+1] == 1 or G[0][j+1] == 1
    elif i == n-1:
        return G[n-1][j-1] == 1 or G[n-2][j-1] == 1 or G[n-2][j] == 1 \
               or G[n-2][j+1] == 1 or G[n-1][j+1] == 1
    elif j == 0:
        return G[i-1][0] == 1 or G[i-1][1] == 1 or G[i][1] == 1 \
               or G[i+1][1] == 1 or G[i+1][0] == 1
    elif j == n-1:
        return G[i-1][n-1] == 1 or G[i-1][n-2] == 1 or G[i][n-2] == 1\
               or G[i+1][n-2] == 1 or G[i+1][n-1] == 1
    else:
        return G[i-1][j-1] == 1 or G[i-1][j] == 1 or G[i-1][j+1] == 1 \
               or G[i][j-1] == 1 or G[i][j+1] == 1 or G[i+1][j-1]== 1 \
               or G[i+1][j] == 1 or G[i+1][j+1] == 1

def bernoulli(p):
    x = rd.random()
    if x <= p:
        return 1
    else:
        return 0

def suivant(G, p1 , p2):
    n = len(G)
    G1 = deepcopy(G)
    # une copie simple ne suffit pas à cause des listes imbriquées!
    for i in range(n):
        for j in range(n):
            if G[i][j] == 0: # case saine
                if est_exposee(G,i,j):
                    G1[i][j] = bernoulli(p2)
            elif G[i][j] == 1: # infectée
                G1[i][j] = 2 + bernoulli(p1)
            # si rétabli ou mort on ne fait rien
    return G1

def suivant_ameliore(G, p1 , p2, nb):
    # amélioration: on met en même temps à jour le décompte
    n = len(G)
    G1 = deepcopy(G)
    # une copie simple ne suffit pas à cause des listes imbriquées!
    for i in range(n):
        for j in range (n):
            if G[i][j] == 0: # case saine
                if bernoulli(p2) == 1 and est_exposee_amelioree(G,i,j):
                    G1[i][j] = 1
                    nb[0] -= 1
                    nb[1] += 1
```

```
            elif G[i][j] == 1: # infectée
                nb[1] -= 1
                if bernoulli(p1) == 1:
                    G1[i][j] = 3
                    nb[3] += 1
                else:
                    G1[i][j] = 2
                    nb[2] +=1
            # si rétabli ou mort on ne fait rien
    return G1

def simulation (n, p1 ,p2):
    # On remarque que tant qu'il y a une personne infectée, la grille évolue
    G = init(n)
    nb = compte(G)
    while nb[1] > 0:
        G = suivant(G, p1 ,p2)
        nb = compte(G)
    return [ nb[i]/n**2 for i in range (4) ]

def simulation_amelioree (n, p1 ,p2):
    # On initialise la grille directement avec une proportion I0 de personnes infectées
    # On ne recalcule plus le décompte à chaque fois, on le met à jour au fur et à mesure
    G = init_multiple(n)
    nb = compte(G)
    while nb[1] > 0:
        G = suivant_ameliore(G, p1 ,p2, nb)
    return [ nb[i]/n**2 for i in range (4) ]

def seuil(Lp2, Lxa):
    imin = 0
    imax = len(Lp2)-1
    while imax-imin > 1:
        m =(imin + imax)//2
        if Lxa[m] < 0.5:
            imin = m
        else :
            imax = m
    return [Lp2[imin], Lp2[imax]]

p1 = 0.2 # 20% de mortalité
I0 = 0.05 # proportion initiale d'individus infectés
n = 1000
Lp2 = np.linspace(0, 1, 51) # p2 évolue, par pas de 0,02%
Lxa = []
for p2 in Lp2:
    print(p2)  # affichage pour patienter!
    S = simulation_amelioree(n, p1, p2)
    Lxa.append(S[2]+S[3])

print("Le seuil critique de pandémie se situe dans l'intervalle", seuil(Lp2,Lxa))

plt.figure(4)
plt.plot(Lp2, Lxa)
plt.title("Proportion de la population atteinte")
plt.xlabel("p2")
plt.ylabel("Proportion de population")

# Bonus: la proportion de morts/rétablis etc...
# comme dans le modèle à compartiments
```

```
p2 = 0.34 # choix arbitraire, pour comparer facilement aux premières courbes
plt.figure(5)
Tmax = 25
T = list(range(Tmax))
G = init_multiple(n)
nb = compte(G)
Nombre = []
for i in T:
    print(i) # pour patienter....
    Nombre.append( [nb[i]/n**2 for i in range(4)] )
    G = suivant_ameliore(G, p1 ,p2, nb)

plt.plot(T, [X[0] for X in Nombre], label="sains")
plt.plot(T, [X[1] for X in Nombre], label="infectés")
plt.plot(T, [X[2] for X in Nombre], label="rétablis")
plt.plot(T, [X[3] for X in Nombre], label="morts")

plt.title("propagation d'une épidémie, méthode des automates cellulaires")
plt.xlabel("Temps")
plt.ylabel("Proportion de population")
plt.legend()

plt.show()
```






❏ **Q25.** On ne peut pas supprimer le test ligne 8 pour plusieurs raisons :

- si jamais au départ on vaccinait les personnes qui ont été infectées lors de l'initialisation, il ne se passerait plus rien !
- il ne faut pas vacciner les cases qui l'ont déjà été, sinon on n'obtiendrait pas exactement la proportion $q$.
- il ne faut pas vacciner les morts...

❏ **Q26.** `init_vac(5, 0.2)` renvoie une grille de taille $5 \times 5$ dont exactement 5 cases valent 2 (vaccinées), une autre vaut 1 (infectée) et toutes les autres valent 0.