DS1 MPSI ITC

Solution. □

Aucun appareil électronique (montre, smartphone, calculatrice...) n'est autorisé.

1 Tri insertion

Question 1.

On s'intéresse au *tri insertion* : un tri *en place* (pas d'utilisation de tableau auxiliaire) et *stable* (pas de modification de l'ordre des *items* égaux).

1. Écrire une fonction `placer(liste:list, i:int)->None` qui place `liste[i]` à sa bonne position dans `liste[:i+1]`, en supposant que `liste[:i]` est déjà triée.

```
t = [10,15,20,25,12,60,23]
placer(t,4)
print(t)
```

```
[10, 12, 15, 20, 25, 60, 23]
```

   La liste passée en paramètre subit des *effets de bords*.

2. Étudier la complexité de l'appel `placer(liste,i)` dans le meilleur cas et dans le pire en fonction des paramètres qui vous semblent pertinents.
3. Le *tri insertion* applique itérativement la fonction de placement. Écrire la fonction `tri_insertion(liste:list)->None` qui trie *en place* une liste.
4. Étudier la complexité de ce tri dans le meilleur cas et dans le pire en fonction de n, taille de la liste.
5. Qu'est-ce qui garanti le caractère *stable* de ce tri dans votre code ?

Solution. Voici les codes :

```
#Q1.1
def placer(liste, i):
    """Place liste[i] à sa bonne position dans liste[:i+1],
    en supposant que liste[:i] est déjà triée."""
    x = liste[i]
    j = i
    # On décale les éléments plus grands vers la droite
    while j > 0 and liste[j - 1] > x:
        liste[j] = liste[j - 1]
        j -= 1
    liste[j] = x

#Q1.3
def tri_insertion(liste):
    """Trie la liste par ordre croissant en utilisant placer."""
    for i in range(1, len(liste)):
        placer(liste, i)
```

Q1.2a et Q1.2b

La première fonction a un meilleur cas en $O(1)$: cela arrive si `liste[i]` est plus grand que `liste[i-1]`.

Elle a un pire cas en $O(i)$: cela arrive si `liste[i]` est plus petit que tous les éléments de `liste[:i]`.

Q1.4a et Q1.4b

Le tri lui-même a un meilleur cas lorsque la liste est déjà triée par ordre croissant : chaque appel à `placer` est en $O(1)$ puisque l'élément courant que celui sur sa gauche.

Le pire cas arrive lorsque chaque élément doit être inséré au début de la liste. L'appel `placer(liste,i)` est en $O(i)$ pour chaque i. Ce cas se produit si la liste est triée dans l'ordre décroissant. La complexité totale est de l'ordre de la somme des premiers entiers jusqu'à n (si n est la longueur de la liste). Elle est majorée par un multiple de

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

Q1.5

L'aspect *stable* est garanti par la ligne `while j > 0 and liste[j - 1] > x:` du placement : l'écriture ne se fait que si l'élément de gauche est STRICTEMENT plus grand que l'élément de droite. Autement dit, si ces deux éléments sont égaux, leur placement relatif e est préservé.

□

2 Tri casier

Idée générale : Le tri casier (ou *counting sort*) repose sur le comptage des occurrences de chaque valeur. On suppose que les éléments à trier sont des entiers compris entre un minimum m et un maximum M.

Étape 1 : déterminer les bornes On parcourt la liste pour trouver le plus petit et le plus grand élément. Ces valeurs m et M permettent de connaître le nombre de cases nécessaires.

Étape 2 : compter les occurrences On crée une liste de compteurs t de taille $M - m + 1$, initialisée à zéro. Pour chaque élément x de la liste, on incrémente $t[x - m]$.

Étape 3 : reconstruire la liste triée On parcourt les indices i de la liste des compteurs : pour chaque i, on ajoute $t[i]$ fois la valeur $i + m$ dans la liste résultat.

Résultat : La liste obtenue contient les mêmes éléments que la liste initiale, mais rangés dans l'ordre croissant.

Question 2.

On implémente le tri casier :

1. Écrire la fonction `minmax(l)` qui renvoie le tuple m, M formé du minimum et du maximum de ℓ. Les fonctions `min` et `max` sont proscrites.
2. Écrire la fonction `compter(l)` qui réalise l'étape 2.
3. Écrire la fonction `trier(l)` qui réalise l'étape 3.
4. Donner la complexité au pire des appels `minmax(l)`, `compter(l)`, `trier(l)` si la liste passée en argument est de taille n que son minimum vaut m et son maximum vaut M.

Solution. Q2.1 ; Q2.2 ; Q2.3

```
1 def minmax(l):
2     """Renvoie un couple (m, M) avec le minimum et le maximum de la liste l."""
3     m = l[0]
4     M = l[0]
5     for x in l:
6         if x < m:
7             m = x
8         if x > M:
```

```
            M = x
    return m, M


def compter(l):
    """Renvoie la liste des occurrences de chaque valeur entre min et max."""
    m, M = minmax(l)
    t = [0] * (M - m + 1)
    for x in l:
        t[x - m] += 1
    return t, m, M


def trier(l):
    """Renvoie une nouvelle liste triée par la méthode du tri casier (counting sort)."""
    t, m, M = compter(l)
    res = []
    for i in range(len(t)):
        for _ in range(t[i])
            res.append(i + m)
    return res
```

Complexité si `l` est de longueur n

- Q2.4a `minmax(l)` est en $O(n)$
- Q2.4b `compter(l)` est en $O(n) + O(M-m)$
- Q2.4 c `trier(l)` coûte $O(n) + O(M-m)$ (appel à `compter` et `minmax`).

  Puis un coût de $O(M-m)$ pour incrémenter les indices `i` dans la boucle externe.

  On ajoute par `append` AU TOTAL autant d'éléments que la somme des `t[i]`. Or cette somme vaut n. Le coût cumulé de tous les `append` est donc un $O(n)$
- Au final $O(n) + O(M-m) + O(n) = O(n + (M-m))$

□

3 Modélisation des épidémies

L'étude de la propagation des épidémies joue un rôle important dans les politiques de santé publique. Les modèles mathématiques ont permis de comprendre pourquoi il a été possible d'éradiquer la variole à la fin des années 1970 et pourquoi il est plus difficile d'éradiquer l'apparition d'épidémies de grippe tous les hivers. Aujourd'hui, des modèles de plus en plus complexes et puissants sont développés pour prédire la propagation d'épidémies à l'échelle planétaire telles que le SRAS, le virus H5N1 ou le virus Ebola. Ces prédictions sont utilisées par les organisations internationales pour établir des stratégies de prévention et d'intervention.

Le travail sur ces modèles mathématiques s'articule autour de trois thèmes principaux : traitement de base des données, simulation numérique (par plusieurs types de méthodes), identification des paramètres intervenant dans les modèles à partir de données expérimentales. Seul le troisième thème est abordé dans ce sujet.

Dans tout le problème, on peut utiliser une fonction traitée précédemment. On suppose que la bibliothèque `random` a été importées par :

```
import random as rd
```

On s'intéresse ici à une méthode de simulation numérique (dite *par automates cellulaires*).

Dans ce qui suit, on appelle grille de taille $n \times n$ une liste de n listes de longueur n, où n est un entier strictement positif.

Pour mieux prendre en compte la dépendance spatiale de la contagion, il est possible de simuler la propagation d'une épidémie à l'aide d'une grille. Chaque case de la grille peut être dans un des quatre états suivants : saine, infectée, rétablie, décédée. On choisit de représenter ces quatre états par les entiers :

0 (Sain), 1 (Infecté), 2 (Rétabli) et 3 (Décédé)

L'état des cases d'une grille évolue au cours du temps selon des règles simples. On considère un modèle où l'état d'une case à l'instant $t+1$ ne dépend que de son état à l'instant t et de l'état de ses huit cases voisines à l'instant t (une case du bord n'a que 5 cases voisines et trois pour une case d'un coin). Les *règles de transition* sont les suivantes :

— une case décédée reste décédée ;

— une case infectée devient décédée avec une probabilité p_1 ou rétablie avec une probabilité $(1-p_1)$;

— une case rétablie reste rétablie ;

— une case saine devient infectée avec une probabilité p_2 si elle a au moins une case voisine infectée et reste saine sinon.

On initialise toutes les cases dans l'état sain, sauf une case choisie au hasard dans l'état infecté.

Question 3. Création de la grille

Écrire une fonction `grille(n:int)->[[int]]` qui renvoie une grille $n \times n$ de zéros (donc une grille de personnes saines).

Solution. Code Q3

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

□

On peut dans la question suivante utiliser la fonction `rd.randrange(p)` de la bibliothèque `random` qui, pour un entier positif p, renvoie un entier choisi aléatoirement entre 0 et $p-1$ inclus.

Question 4. Initialisation aléatoire

Écrire en Python une fonction `init(n:int)->[[int]]` qui choisit une case aléatoire saine, la transforme en case infectée, puis renvoie une grille $n \times n$ avec une seule personne infectée.

Solution. Code Q4

```
def init(n):
    g = grille(n)
    #ligne, colonne
    i,j=randrange(n),randrange(n)
    g[i][j]=1#case infectée
    return g
```

□

Question 5. Compter les états

Écrire en Python une fonction `compte(G)` qui renvoie une liste formée des nombres de cases dans chacun des quatre états.

Solution. Code Q5

```
def compte(G):
    r=[0]*4
    for i in range(len(G)):
        for j in range(len(G)):
            r[G[i][j]]+=1
    return r
```

□

D'après les règles de transition, pour savoir si une case saine peut devenir infectée à l'instant suivant, il faut déterminer si elle est exposée à la maladie, c'est-à-dire si elle possède au moins une case infectée dans son voisinage. Pour cela, on écrit en Python la fonction `est_exposee(G,i,j)` suivante :

```
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:
                # a completer
        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) == 0
        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) == 0
        else:
                # a completer
```

Question 6. À propos de `est_exposee`

On n'écrit sur la copie que le type de la fonction et les lignes 12 et 23 :

1. Quel est le type du résultat renvoyé par la fonction `est_exposee(G,i,j)` ?
2. Compléter les lignes 12 et 23 de la fonction `est_exposee(G,i,j)`.

Solution. Le type de retour est `bool`. Q6.1

```
    #L12 Q6.2a
    return ( G [0][j-1] -1)*( G [0][ j+1] -1)*\
         ( G [1][j-1] -1)*( G [1][j] -1)*( G [1][ j+1] -1)== 0

    #L23 Q6.2b
```

```
    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
```

□

La fonction `random()` renvoie un flottant aléatoire choisit selon une loi de distribution uniforme dans $[0.0; 1.0[$. On veut l'utiliser pour modéliser le comportement d'une *variable aléatoire suivant une loi de Bernoulli*. Pour $p \in [0; 1[$ une telle variable renvoie Vrai avec une probabilité de p et Faux avec une probabilité de $1 - p$.

Question 7. Loi de Bernoulli

Écrire une fonction `bernoulli(p:float)->bool` qui renvoie `True` avec une probabilité de p. Si $p > 1$, la fonction renvoie toujours `True`, et toujours `False` pour $p < 0$.

Solution. Code Q7

```
def bernoulli(p: float) -> bool:
    """Renvoie True avec la probabilité p, False sinon."""
    return rd.random() < p
```

□

Question 8. Copie d'une grille

Écrire la fonction `copy(g:[[int]])->[[int]]` qui renvoie une copie de la grille passée en argument.

Solution. Code Q8

```
    def copy(g:[[int]]):
    n = len(g)
    ng = grille(n)
    for i in range(n):
        for j in range(n):
            ng[i][j]=g[i][j]
    return ng
```

□

Question 9. Transition d'un pas

Écrire une fonction `suivant(G:[[int]],p1:float,p2:float)->bool` qui met à jour la grille `G`. Au moment de l'appel à la fonction, `G` représente la situation en un temps t, après l'exécution la situation reprsentée est celle du temps $t + 1$. La fonction renvoie un bouléen qui est faux si et seulement si aucun changement n'a été effectué dans G.

Solution. Code

```
def suivant(G,p1,p2):
    c=0#compteur de modifications
    g=copy(G)#g représente la situation au temps t; G au temps t+1
    for i in range(len(G)):
        for j in range(len(G)):
            if g[i][j]==0 and est_exposee(g,i,j):
                #case saine infectable
                if bernoulli(p2):#case devient infectée
                    G[i][j]=1
```

```
                    c+=1
            elif g[i][j]==1:#case infectée
                if bernoulli(p1):
                    #décès
                    G[i][j]=3
                    c+=1
                else:
                    #rétablissement
                    G[i][j]=2
                    c+=1
            else:
                pass
    return c > 0
```

□

Avec les règles de transition du modèle utilisé, l'état de la grille évolue entre les instants t et $t+1$ tant qu'il existe au moins une case infectée.

Question 10. Simulation complète

Écrire en Python une fonction `simulation(n:int,p1:float,p2:float)->[[int]]` qui renvoie les proportions de chaque état à chaque étape (la simulation s'arrête lorsque la grille n'évolue plus).

Solution. Code Q9

```
def simulation(n,p1,p2):
    g=init(n)
    res = []
    while suivant(g,p1,p2):
        res.append([e/n**2 for e in compte(g)])
    return res
```

On récupère la liste des proportions après chaque étape. La dernière proportion est celle qui correspond à une grille stable (plus d'infectés). □

Question 11. Proportion finale d'infectés

Quelle est la valeur de la proportion des cases infectées `x1` à la fin d'une simulation ? Quelle relation vérifient `x0,x1,x2` et `x3` ? Comment obtenir à l'aide des valeurs de `x0,x1,x2` et `x3` la valeur `x_atteinte` de la proportion des cases qui ont été atteintes par la maladie pendant une simulation ?

Solution. Q11a ; Q11b et Q11c

Il n'y a plus de case infectée, donc $x_1 = 0$

On a $x_0 + x_2 + x_3 = 1$

Les personnes qui ont été atteintes sont rétablies ou mortes. La proportion des cases qui ont été atteintes par la maladie pendant une simulation est $x_2 + x_3$ ou $1 - x_0$ ce qui revient au même □

On fixe `p1` à 0,5 et on calcule la moyenne des résultats de plusieurs simulations pour différentes valeurs de `p2`. On obtient la courbe de la figure 1.

On appelle *seuil critique de pandémie* la valeur de `p2` à partir de laquelle plus de la moitié de la population a été atteinte par la maladie à la fin de la simulation. On suppose que les valeurs de `p2` et `x_atteinte` utilisées pour tracer la courbe de la figure 1 on été stockées dans deux tableaux de même longueur k : `Lp2` (valeurs en abscisses dans le graphique 1) et `Lxa` (valeurs en ordonnées dans le graphique 1). Ces deux tableaux sont triés par ordre croissant et `Lxa[i]` désigne le nombre de personnes atteintes calculées par la simulation pour la probabilité `Lp2[i]`.

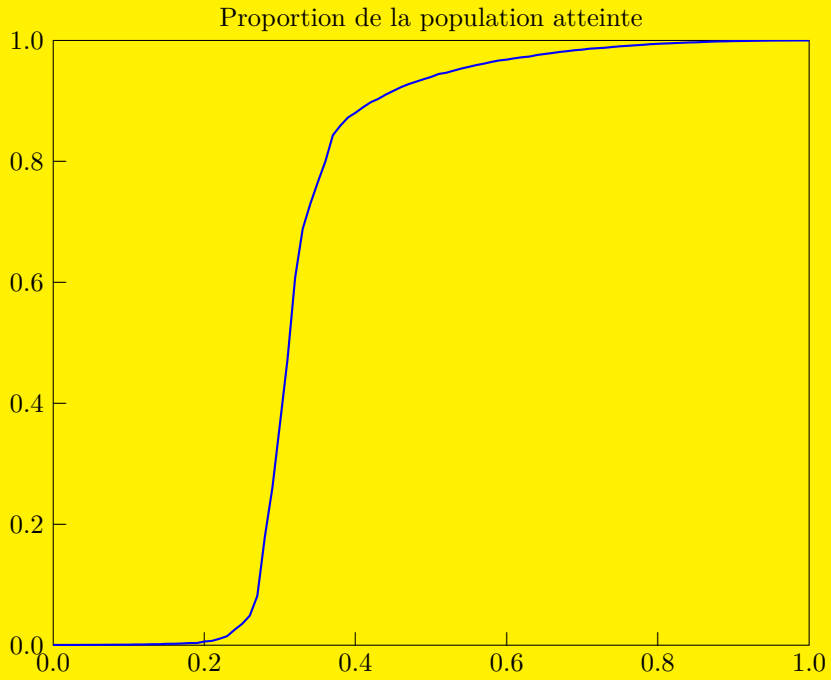

FIGURE 1 – Représentation de la proportion de la population qui a été atteinte par la maladie pendant la simulation en fonction de la probabilité p_2 si p_1 est fixé à 0.5

Question 12. Recherche du seuil critique

Écrire une fonction `seuil(Lp2:[[float]],Lxa[[float]])->(float,float)` qui détermine un encadrement $[m, M]$ du seuil critique de pandémie avec la plus grande précision possible ($|M - m|$ doit être minimum). Le retour de la fonction est donc un tableau dynamique qui contient 2 valeurs.

On suppose que la liste `Lp2` croît de 0 à 1 et que la liste `Lxa` des valeurs correspondantes est croissante.

Une complexité logarithmique en k est attendue.

Solution. Q12

Il s'agit de faire une recherche dichotomique

```
def seuil(Lp2,Lxa)->(float,float):#fonction demandée
    g, d = 0, len(Lxa)-1
    while d - g >1:#invariant [Lp2[g]<= 0.5 <= Lp2[d]
        m = int((g+d)/2)
        if Lxa[m] == 0.5:
            return [Lp2[m],Lp2[m+1]]
        elif Lxa[m] < 0.5:
            g = m
        else:
            d = m
    return [Lp2[g],Lp2[d]]
```

□

Pour étudier l'effet d'une campagne de vaccination, on immunise au hasard à l'instant initial une fraction `q` de la population. On a écrit la fonction `init_vac(n,q)`.

```
1 def init_vac(n,q):
2         G = init(n)
3         nvac = int(q*n**2)
4         k = 0
5         while k < nvac:
6                 i = rd.randrange(n)
7                 j = rd.randrange(n)
8                 if G[i][j] == 0:
9                         G[i][j] = 2
10                        k += 1
11        return G
```

Question 13. Test ligne 8

Donner les raisons qui motivent le test en ligne 8.

Solution. Q13

Il y a une personne infectée à l'instant initial, et le vaccin est sans effet sur elle. Il ne faut pas vacciner une personne malade.

De plus, il ne faudrait pas vacciner deux fois la même personne. □

Question 14.

Que renvoie l'appel `init_vac(5,0.2)` ?

Solution. Q14

Une grille 5×5 avec $25\frac{2}{10} = 5$, donc 5 personnes notées comme rétablies (vaccinées) et une malade. □

Appendice

Rappel : slicing en Python

En Python, on peut extraire une partie d'une liste, d'une chaîne ou d'un tuple à l'aide de la notation `seq[d:f:p]`, où :

- `d` est l'indice de **début** (inclus),
- `f` est l'indice de **fin** (exclu),
- `p` est le **pas** (facultatif, par défaut 1).

Si un argument est omis, Python prend la valeur par défaut :

- `d=0` si on omet le début ;
- `f=len(seq)` si on omet la fin ;
- `p=1` si on omet le pas.

Exemples :

- `L = [10, 20, 30, 40, 50, 60]`
- `L[1:4]` renvoie `[20, 30, 40]` (éléments d'indice 1 à 3) ;
- `L[:3]` renvoie `[10, 20, 30]` ;
- `L[3:]` renvoie `[40, 50, 60]` ;
- `L[::-1]` renvoie `[60, 50, 40, 30, 20, 10]` (liste renversée) ;
- `'informatique'[::2]` renvoie `'ifraiue'` (une lettre sur deux).