Exercice I

sujet des Mines 2016 sans la base de données

I Partie I : tri

☐ **Q1 -** On déroule l'appel de la fonction `tri([5,2,3,1,4])`

étape n°	i	L
1		[5,2,3,1,4]
8	1	[2,5,3,1,4]
18	2	[2,3,5,1,4]
31	3	[1,2,3,5,4]
47	4	[1,2,3,4,5]

☐ **Q2 -** On effectue une démonstration par récurrence.

Soit H_i : "la liste L[0 :i+1] est triée par ordre croissant à l'issue de l'itération i"

initialisation : H_0 est vraie car la liste L[0 :0+1] correspond alors au seul premier élément qui est nécessairement trié.

supposons H_{i-1} vérifiée : dans la liste L[0 :i+1], la sous-liste L[0 :i] est triée. Seule la valeur finale L[i] n'est pas nécessairement triée.

j prend la valeur i et x la valeur L[i]. La boucle `while` vérifie si la dernière valeur de la sous liste triée L[0 :i] (L[j]) est plus grande que x, auquel cas cette valeur remonte d'une case dans la liste et j décroit de 1 afin de vérifier la prochaine valeur. On remarque que cette dernière valeur L[j] a été copiée et se retrouve 2 fois dans la liste côte à côte.

Dès que L[j] est plus petit que x, on a trouvé la place de x et on l'y met en écrasant la 2^e valeur L[j].

Ainsi en sortie de la boucle `while`, L[0 :i+1] est triée et H_i est vraie

conclusion : H_i est vraie pour tout $i \geqslant 0$

En particulier, à la sortie de la boucle `for` i=n, donc H_n est vraie et toute la liste L est triée.

☐ **Q3 -** Le meilleur des cas est la liste déjà triée (*e.g.* [1,2,3,4,5]). Le coût temporel est alors $\mathcal{O}(n)$

Le pire des cas est la liste triée à l'envers (*e.g.* [5,4,3,2,1]) puisque la boucle `while` fera à chaque fois i itérations. Il y aura donc $1+2+3+\ldots n = \frac{n(n+1)}{2}$ itérations de boucles.

Le coût temporel est alors $\mathcal{O}(n^2)$

On peut penser à un tri fusion qui aura un coût en $\mathcal{O}(n\ ln(n))$ quel que soit les cas.

☐ **Q4 -** On propose la modification suivante :

```
def tri_chaine(L):
    n = len(L)
    for i in range(1,n):
        j = i
        x = L[i]        # on garde les deux elements, expl : ['toto',24]
        while 0 < j and x[1] < L[j-1][1]:
            L[j] = L[j-1]
            j = j-1
        L[j] = x
```

II Partie II : Modèle à compartiments

☐ **Q5 -** On pose $X = \begin{pmatrix} S \\ I \\ R \\ D \end{pmatrix}$

$$f : (\mathscr{C}^1(\mathbb{R}+))^4 \to (\mathscr{C}^1(\mathbb{R}+))^4$$

et

$$\begin{pmatrix} S \\ I \\ R \\ D \end{pmatrix} \mapsto \begin{pmatrix} t \mapsto -rS(t)I(t) \\ t \mapsto rS(t)I(t) - (a+b)I(t) \\ t \mapsto aI(t)) \\ t \mapsto bI(t)) \end{pmatrix}$$

On a bien alors $\frac{d}{dt}X = f(X)$

☐ **Q6 -** On complète par :

```
def f(X):
    """ Fonction definissant l'equation differentielle """
    global r, a, b
    L=[]
    L.append(-r*X[0]*X[1])
    L.append(-L[0]-(a+b)*X[1])
    L.append(a*X[1])
    L.append(b*X[1])
    return np.array(L)
```

☐ **Q7 -** Les deux simulations consistent à évaluer pour intervalle entre 0 et 25 unités de temps l'évolutions des quatre catégories.

La première décide d'évaluer cette évolution toutes les $\frac{25}{7} \approx 3{,}57$ unités de temps. La seconde évalue cette évolution toutes les $\frac{25}{250} = 0{,}1$ unités de temps.

La seconde devrait donc être plus précise mais requiert plus de calculs (250 évalutions de $f(X)$ au lieu de 7 pour la première).

☐ **Q8 -** On complète ainsi :

```
def f(X, Itau):
    """ Fonction definissant l'equation differentielle
    Itau est la valeur de I(t - p * dt ) """
    global r, a, b
    L=[]
    L.append(-r*X[0]*Itau)
    L.append(-L[0]-(a+b)*X[1])
    L.append(a*X[1])
    L.append(b*X[1])
    return np.array(L)

# sans changement

# Methode d'Euler
for i in range(N):
    t = t + dt
    if i < p :
        Itau = X0[1]
    else :
        Itau = XX[i-p][1]
    X = X + dt * f(X, Itau)
    tt.append(t)
    XX.append(X)
```

☐ **Q9 -** En évaluant l'intégrale, la seule modification par rapport à la relation de la question précédente est le calcul de la valeur de `Itau`.

Ainsi sans modifier la fonction `f`, on peut apporter les modifications suivantes :

```
# sans changement

# Methode d'Euler
for i in range(N):
    t = t + dt
```

```
    # debut des modifications
    Itau = 0
    for j in range(p):
        if i < p :
            Itau += X0[1]*h(dt*j)
        else :
            Itau += XX[i-j][1]*h(dt*j)
    Itau = dt*Itau
    # fin des modifications
    X = X + dt * f(X, Itau)
    tt.append(t)
    XX.append(X)
```

III Partie III : Modélisation dans des grilles

☐ **Q10 -** La fonction grille renvoie la matrice nulle de $\mathscr{M}_n(\mathbb{R})$.

☐ **Q11 -** On peut suggérer :

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

☐ **Q12 -** On écrit :

```
def compte(G):
    n = len(G)
    L=[0,0,0,0]
    for i in range(n):
        for j in range(n):
            L[ G[i][j] ] += 1     # G contient les 4 etats : 0, 1, 2, 3
    return np.array(L)            # pour preparer la fonction simulation
```

☐ **Q13 -** La fonction `est_exposee` renvoie un booléen.

☐ **Q14 -** On complète ainsi :

```
def est_exposee(G, i, j):
    # non modifie
    elif i == 0:
        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
    # non modifie
    else: # on est au milieu de la grille, il y a 8 voisins
        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
```

☐ **Q15 -** En appliquant les règles de transition, on a :

```
def suivant(G, p1, p2):
    n = len(G)
    GG = grille(n)
    for i in range(n):
        for j in range(n):
            if G[i][j] == 0 and est_expose(G,i,j) :
                GG[i][j] = bernoulli(p2)
            elif G[i][j] == 1 :
```

```
            GG[i][j] = 2 + bernoulli(p1)
         else :
            GG[i][j] = G[i][j]
   return GG
```

☐ **Q16 -** En s'inspirant de la partie II, on peut écrire :

```
def simulation(n, p1, p2):
    G = init(n)
    L = compte(G)
    while L > 0 :    # il n'y a plus de personnes infectees
        G = suivant(G,p1,p2)
        L = compte(G)
   return L / n**2   # L est un np.array, sinon [x/n**2 for x in L]
```

☐ **Q17 -** À la fin d'une simulation $x1 = 0$. Les individus infectés sont soit immunisés soit morts.

On a la relation $x1 + x2 + x3 + x4 = 1$.

Pour obtenir la valeur des individus atteints, on calcule $x\_atteinte = x_1 + x_2 + x_3 = 1 - x_0$

☐ **Q18 -** Pour encadrer par dichotomie le seuil critique, on peut écrire :

```
def seuil(Lp2, Lxa):
   n = len(Lp2)
   a = 0
   b = n
   while b-a > 1 :
      c = (a+b)//2      # division euclidienne
      if Lxa[c] > 0.5 :
         b = c
      else :
         a = c
   return [ Lp2[a],Lp2[b] ]
```

☐ **Q19 -** Le test de la ligne 8 permet de ne pas vacciner des individus infectés. Si on retire donc le test, on pourrait se retrouver avec une matrice sans individu infecté. Il n'y aura donc pas d'évolution au cours du temps, ce qui retirerait l'intérêt de la simulation. On gardera donc le test.

☐ **Q20 -** L'appel `init_vac(5, 0.2)` renvoie une matrice 5×5 avec un élément égal à 1 (individu infecté), cinq éléments ($25 \times 0{,}2$) égaux à 2 (individus vaccinés) et le reste nul (individus sains et non vaccinés).

Exercice II

Au cours du développement des fonctions nécessaires à la manipulation des nombres premiers on s'aperçoit que le choix des algorithmes pour évaluer chaque fonction est primordial pour garantir des performances acceptables. On souhaite donc mener des tests à grande échelle pour évaluer les performances réelles du code qui a été développé. Pour ce faire on effectue un grand nombre de tests sur une multitude d'ordinateurs. Les données sont ensuite centralisées dans une base de données composée de deux tables.

La première table est `ordinateur` et permet de stocker des informations sur les ordinateurs utilisés pour les tests. Ses attributs sont :

- `nom CHAR`, clé primaire, le nom de l'ordinateur.
- `gflops INTEGER`, la puissance de l'ordinateur en milliards d'opérations flottantes par seconde.
- `ram INTEGER`, la quantité de mémoire vive de l'ordinateur en Go.

Exemple du contenu de cette table :

nom	gflops	ram
nyarlathotep114	69	32
nyarlathotep119	137	32
...		
shubniggurath42	133	16
azathoth137	85	8

La seconde table est `fonctions` et stoke les informatioons sur les tests effectués pour différentes fonctions en cours de développement. Ses attributs sont :

- `id INTEGER`, l'identifiant du test effectué.
- `nom CHAR`, le nom de la fonction testée.
- `algorithme CHAR`, le nom de l'algorithme qui permet le calcul de la fonction testée.
- `teste_sur TEXT`, le nom du PC sur lequel le test a été effectué.
- `temps_exec INTEGER`, le temps d'exécution du test en millisecondes.

Exemple du contenu de cette table :

id	nom	algorithme	teste_sur	temps_exec
1	li	rectangle	nyarlathotep165	2638
2	li	rectangle	shubniggurath28	736
3	li	trapezes	nyarlathotep165	4842
...				
2154	Ei	puiseux	nyarlathotep145	2766
2155	aleatoire	bbs	azathoth145	524

Q1. L'attribut `nom` ne possède pas une valeur unique pour chaque entité de la table `fonction` (*cf.* il y a 3 noms égaux à `li` dans l'exemple de contenu), ainsi il ne peut être utilisé comme clef primaire.

Q2. Les informations sont contenues dans la table `ordinateur`,

```
SELECT nom FROM ordinateur
WHERE ram = 32 ;
```

Q3. Les informations sont contenues dans la table `fonction`,

```
SELECT teste_sur, nom FROM fonction ;
```

Q4. Les informations sont contenues dans la table `fonction`,

```
SELECT DISTINCT algorithme FROM fonction ;
```

Q5. Les informations sont contenues dans la table `fonction`,

```
SELECT teste_sur FROM fonction
WHERE temps_exec > 1000 AND nom = 'li' ;
```

Q6. Les informations sont principalement contenues dans la table `fonction`, il faut néanmoins joindre les deux tables afin d'obtenir la puissance des ordinateurs.

```
SELECT algorithme, teste_sur, ordi.gflops FROM fonction
INNER JOIN ordinateur AS ordi ON ordi.nom = teste_sur
WHERE nom = 'Ei'
ORDER BY temps_exec DESC
```

Rq : vous pouvez simplement écrire `JOIN` plutôt que `INNER JOIN`.

Q7. Les informations sont contenues dans la table `fonction`. On exclut les ordinateurs à l'aide d'une sous-requête qui renvoie tous les ordinateurs testés pour la fonction `li`.

```
SELECT teste_sur FROM fonction
WHERE teste_sur NOT IN ( SELECT teste_sur FROM fonction
                           WHERE nom = 'li'
                         ) ;
```

Q8. Les informations sont contenues dans la table `fonction` (on prendra garde à ne pas compter de doublons).

```
SELECT COUNT(DISTINCT teste_sur) FROM fonction
WHERE nom = 'li' ;
```

Q9. Les informations sont contenues dans la table `fonction`, il faudra regrouper les moyennes selon les noms des fonctions.

```
SELECT nom, AVG(temps_exec) FROM fonction
GROUP BY nom
```

Q10. Les informations sont contenues dans la table `fonction`. On obtient la moyenne d'exécution pour la fonction `li` à l'aide d'une sous-requête.

```
SELECT teste_sur FROM fonction
WHERE nom = 'li'
  AND temps_exec >= ( SELECT AVG(temps_exec) FROM fonction
                        WHERE nom = 'li'
                      ) ;
```

Q11. La sous-sous-requête renvoie le nombre (`nbr`) de fonctions testées sur chaque ordinateur. Ainsi la sous-requête renvoie le nombre maximal de fonctions testées sur un ordinateur.

De là, la requête renvoie le nom des ordinateurs qui ont fait le plus grand nombre de tests.