DS 02 - Python - Correction

On fera vraiment attention à la syntaxe, en particulier :

- Bien utiliser `==` pour les tests d'égalité.
- L'indentation (blocs après `def`, `for`, `while`, `if`, `elif`, `else`, ...) et les deux-points en fin de ligne.
- Les parenthèses pour les appels de fonction (`range(...)`)

On rappelle que `range(a, b)` renvoie la liste d'entiers compris entre `a` inclus et `b` exclu. En particulier `range(len(t))` renvoie bien tous les indices d'une liste `t` donnée.

1 Syntaxe et exemples classiques

1. Ecrire la fonction `factorielle(n)` de deux manières différentes : itérative, puis récursive.

```
def factorielle_iterative(n):
    p = 1
    for i in range(2, n+1): # de 2 (ou 1) à n inclus
        p = p * i
    return p

def factorielle_recursive(n):
    if n == 0:
        return 1
    else:
        return n * factorielle_recursive(n-1)
```

2. Écrire une fonction `somme(t1, t2)` qui prend en entrée deux listes (pas forcément de même taille !) et renvoie une *nouvelle* liste : la somme terme à terme.

```
>>> somme([1, 2, 3], [1, 1, 1, 1, 1, 2, 2, 2, 2])
[2, 3, 4, 1, 1, 2, 2, 2, 2]
```

```
def somme(t1, t2):
    res = [0]*max(len(t1), len(t2))
    for i in range(len(t1)):
        res[i] += t1[i]
    for i in range(len(t2)):
        res[i] += t2[i]
    return res
```

3. Ecrire une fonction `dernier_au_dessus_seuil(t, s)` qui prend en entrée une liste d'entiers `t` et un entier `s` et renvoie la position (indice) de la dernière valeur de `t` strictement plus grande que `s`. Ou `None` s'il n'en existe pas. Sans justifier : quelle est la complexité de votre fonction ?

Ecrire une autre fonction `premier_au_dessus_seuil(t, s)` qui renvoie cette fois la position de la première occurrence : quelle est la complexité dans le pire cas ? le meilleur des cas ?

```
>>> dernier_au_dessus_seuil([1, 7, 2, 8, 4], 5) # renvoie l'indice de 8
```



```
3
>>> premier_au_dessus_seuil([1, 7, 2, 8, 4], 5) # renvoie l'indice de 7
1
```

```
def dernier_au_dessus_seuil(t, s): # Rq : on peut faire plus efficace,
    d = None                       # en utilisant la méthode ci-dessous
    for i in range(len(t)):    # on a besoin des indices
        if t[i] > s:           # on parcourt toute la liste
            d = i              #    au cas où il y en aurait d'autres
    return d
```

Complexité : $\Theta(n)$

```
def premier_au_dessus_seuil(t, s):
    for i in range(len(t)):
        if t[i] > s:
            return i           # On peut s'arrêter tout de suite
    return None
```

Complexité pire cas : $O(n)$. Meilleur cas : $O(1)$

2 Complexités

4. Soit $C(n)$ vérifiant : $C(n) = 2 \times C(\frac{n}{2}) + n, \forall n > 1$, et $C(1) = 0$

Avec $n = 2^k$, montrer par récurrence (rédiger **proprement** !) que $\forall k \geq 0 : C(2^k) = 2^k \times k$. Je ne veux pas voir de n ou de $\mathcal{O}(...)$ dans votre réponse.

Donner un exemple d'algorithme (juste le nom hein) dont la complexité vérifie cette relation.

La relation donne : $C(2^k) = 2 \times C(2^{k-1} + 2^k)$. On montre pas récurrence que $C(2^k) = 2^k \times k$.

- Initialisation ($k = 0$) : $C(2^0) = 0 = 2^0 \times 0$.
- Hérédité ($k \geq 1$ fixé) : On suppose $(HR) : C(2^k) = 2^k \times k$.
  - On a alors : $C(2^{k+1}) \overset{(rel)}{=} 2 \times C(2^k) + 2^{k+1} \overset{(HR)}{=} 2 \times 2^k \times k + 2^{k+1} = 2^{k+1} \times (k + 1)$
- Conclusion : OK. (Remarque : c'est là qu'on pourrait conclure que $C(n) = \Theta(n.log_2(n))$)

On retrouve cette complexité pour le tri fusion.

5. Soit $C(n)$ vérifiant : $C(n) = C(\frac{n}{2}) + 1, \forall n > 1$ et $C(1) = 1$.

Avec $n = 2^k$, montrer que $C(2^k) = k + 1$. Donner un exemple d'algorithme dont la complexité vérifie cette relation.

- Même principe : on a la relation $C(2^k) = C(2^{k-1}) + 1$ et on veut montrer $C(2^k) = k + 1$
- On vérifie pour $k = 0$, puis $C(2^{k+1}) \overset{(rel)}{=} C(2^k) + 1 \overset{(HR)}{=} (k + 1) + 1$.
- C'est la complexité de la dichotomie par exemple : $C(n) = \mathcal{O}(log_2(2))$.

3 Equations différentielles - Susceptible, Infectious, or Recovered

6. On considère dans la suite une modélisation classique de propagation d'épidémie : le modèle SIR représenté par un système d'équations différentielles faisant intervenir trois fonctions (les individus **S**usceptibles d'être infectés ; **I**nfectés ; et en **R**émission) qui dépendent du temps. Les grandeurs α et β sont des constantes qu'on considère déjà définies (on pourra utiliser les variables `alpha` et `beta` directement dans du code Python)

$$\begin{cases} \frac{dS}{dt} = -\beta SI \\ \frac{dI}{dt} = \beta SI - \alpha I \\ \frac{dR}{dt} = \alpha I \end{cases}$$

Montrer que la population totale est considérée dans ce modèle comme constante au cours du temps ;

puis vectorialiser le système précédent pour se ramener à l'ordre 1 et écrire le système sous la forme $Z'(t) = F(Z(t), t)$ où Z et F sont à déterminer.

La population totale est constante car $\frac{d}{dt}(S+I+R) = (-\beta SI) + (\beta SI - \alpha I) + (\alpha I) = 0$.

Vectorialisation : on pose $Z := \begin{pmatrix} S \\ I \\ R \end{pmatrix}$. On a alors

$$Z' = \begin{pmatrix} S' \\ I' \\ R' \end{pmatrix} = F(Z,t) = \begin{pmatrix} -\beta SI \\ \beta SI - \alpha I \\ \alpha I \end{pmatrix}$$

7. Uniquement pendant cette question, on utilise les notations mathématiques :

On considère une fonction y de classe $\mathcal{C}^1$, et on admet que sur un intervalle $[a;b]$ on a $y(b) = y(a) + \int_a^b y'(t)dt$.

Expliquer, sur l'intervalle $[t_k; t_{k+1}]$, quelle approximation on fait pour obtenir la méthode d'Euler, puis donner le shéma d'Euler explicite, en prenant bien soin d'écrire y_k et plus $y(t_k)$ par exemple.

Sur $[t_k; t_{k+1}]$, on approxime $y'(t)$ par $y'(t_k)$ (constante). L'intégration de la constante nous donne :

$$y(t_{k+1}) = y(t_k) + (t_{k+1} - t_k) \times y'(t_k).$$

Comme de plus, on note $y'(t_k) := F(y(t_k), t_k)$ et que l'on considère les approximations $y_k \simeq y(t_k)$, on obtient le schéma d'Euler explicite :

$$y_{k+1} = y_k + (t_{k+1} - t_k) \times F(y_k, t_k).$$

8. Compléter le code suivant (tout recopier svp) : où on manipule un liste Python de petits tableaux numpy.

```
def F(Z, t):
    S, I, R = Z
    return ...

def euler(F, y0, temps):
    res = [np.array(y0)]
    for k in range(len(temps)-1):
        res.append(...)
    return np.array(res)
```

De quelle forme est `y0` ? et le résulat final renvoyé par cette fonction ?


```
def F(Z, t):
    S, I, R = Z
    return -beta*S*I, beta*S*I-alpha*I, alpha*I

def euler(F, y0, temps):
    res = [np.array(y0)]
    for k in range(len(temps)-1):
        res.append(res[-1] + (temsp[k+1]-temps[k])*np.array(F(res[-1], temps[k])))
    return np.array(res)
```

où `y0` est un triplet `(s0, i0, r0)` (ou une liste) et le résultat un `np.array` de n lignes et 3 colonnes.

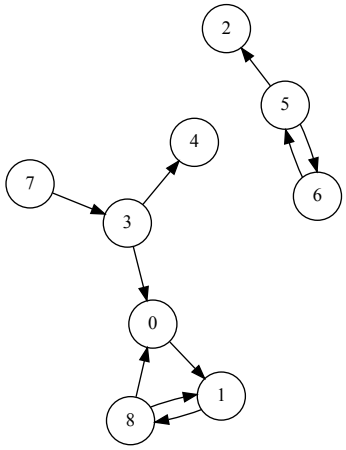
9. On recommence exactement la même chose mais sous une forme un peu différente : on n'utilise plus `F`, mais directement trois listes de taille fixée.

```
def SIR(s0, i0, r0, temps):
   n = len(temps)
   S, I, R = [[0]*n for _ in range(3)] # 3 listes de taille n
   S[0], I[0], R[0] = ...
   for k in range(n-1):
       ...
       S[...] = ...
       I[...] = ...
       R[...] = ...
   return ...
```

```
def SIR(s0, i0, r0, temps):
   n = len(temps)
   S, I, R = [[0]*n for _ in range(3)] # 3 listes de taille n
   S[0], I[0], R[0] = s0, i0, r0
   for k in range(n-1):
       h = temps[k+1]-temps[k]
       S[k+1] = S[k] + h * (-beta*S[k]*I[k])
       I[k+1] = I[k] + h * (beta*S[k]*I[k]-alpha*I[k])
       R[k+1] = R[k] + h * (alpha*I[k])
   return S, I, R
```

4 Parcours de graphe en profondeur

Un graphe est un ensemble de n sommets (ici numérotés de 0 à $n-1$) et d'arcs : s'il existe un arc $i \to j$, on dit que j est un voisin de i. Exemple de graphe :



Pour manipuler un graphe avec python, on peut soit le représenter sous forme de matrice d'adjacence : $M_{i,j} = 1$ si j est un voisin de i, et 0 sinon. C'est une matrice carrée.

```
M = [
  [0, 1, 0, 0, 0, 0, 0, 0, 0],  # voisins de 0
  [0, 0, 0, 0, 0, 0, 0, 0, 1],
  [0, 0, 0, 0, 0, 0, 0, 0, 0],
  [1, 0, 0, 0, 1, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 1, 0, 0, 0, 1, 0, 0],  # les voisins de 5 sont 2 et 6
  [0, 0, 0, 0, 0, 1, 0, 0, 0],
  [0, 0, 0, 1, 0, 0, 0, 0, 0],
  [1, 1, 0, 0, 0, 0, 0, 0, 0],
]
```

soit sous forme de listes d'adjacence : `L` est une liste de n listes, et `L[i]` est la liste des voisins du sommet i. Dans la suite chaque liste de voisin sera ordonnée.

```
L = [
  [1],    # voisins de 0
  [8],
  [],     # liste vide quand il n'y a pas de voisins
  [0, 4],
  [],
  [2, 6], # voisins de 5
  [5],
  [3],
  [0, 1],
]
```


10. Ecrire une fonction `listes_adjacence(M)` qui, étant donnée une matrice d'adjacence d'un graphe, renvoie les listes d'adjacences correpsondantes (Si `M` est de taille $n \times n$, on renverra une liste de n listes)

```
def listes_adjacences(M):
    n = len(M)   # on suppose que M est carrée
    L = [[] for _ in range(n)]
    for i in range(n):
       for j in range(n):
          if M[i][j] == 1:
              L[i].append(j)   # j est voisin de i
    return L
```

Rq : on pourrait aussi créer une nouvelle liste vide à chaque fois, la remplir et la rajouter à `L`.

11. On prend un point de départ (une *source* s) et on s'intéresse à l'ensemble des sommets qui sont accessibles depuis s (i est accessible depuis s s'il est s, ou un voisin de s ou un voisin d'un sommet accessible depuis s.)

Exemple : les sommets accessibles depuis 3 sont $\{0, 1, 3, 4, 8\}$ mais pas 7 (ni 2, 5, 6 hein).

Pour déterminer l'ensemble des sommets accessibles depuis s, on peut faire une parcours en profondeur. Pseudo-code :

```
fonction ParcoursProfondeur(graphe, s)
    accessibles <- {}                   # on utlisera une liste
    pile <- PileVide()                  # ici aussi
    Empiler(pile, s)
    TantQue NonVide(pile):
        u <- ExtrairePile(pile)
        Si Non(u Dans accessibles):     # On explore u
            accessibles <- Union(accessibles, {u})
            PourTout v Dans Voisins[u]: # Les voisins de u dans le graphe, à définir
                Empiler(pile, v)
    Renvoyer accessibles
```

où Empiler(...) correspond à l'opération "push/append" et ExtrairePile à "pop".

Ecrire une fonction python `parcours_profondeur(M, s)` qui correspond : on utilisera des listes Python toutes simples pour `accessibles` et pour la `pile` (on pourra utiliser sans problème `t.append(x)`, `t.pop()`, `t.extend(liste)`, ...) ; `graphe` sera donné sous forme de matrice d'adjacence, et on pourra commencer par le transformer en listes d'adjacence.

```
def parcours_profondeur(M, s):
    voisins = listes_adjacences(M)
    accessibles = []
    pile = []
    pile.append(s)
    while pile != []:
        u = pile.pop()
        if not(u in accessibles):    # ou créer/utiliser appartient(u, accessibles)
            accessibles.append(u)
            for v in voisins[u]:     # ou `accessibles.extend(voisins[u])`
                accessibles.append(v)
    return accessibles
```

On évitera de recalculer plein de fois `listes_adjacences(M)` : 1 fois ça suffit largement !

12. Expliquer à quoi sert la ligne `Si Non(u Dans accessibles):`.

Que se passe-t-il si on remplace la pile par une autre structure de données (une file par exemple) ?

- Cette ligne permet de ne pas avoir de boucle infinie (comme avec 1 et 8 dans l'exemple)
- Si on change de structure de donnée, ça changera l'ordre de parcours (donc l'ordre des éléments dans la liste), mais pas l'ensemble des éléments (donc pas les éléments qui apparaissent dans la liste).

13. Sur le graphe ci-dessus, détailler les étapes pour obtenir le résultat avec $s = 3$.

Exploration	accessibles	pile
		[3]
3	[3]	[0, 4]
4	[3, 4]	[0, 1]
0	[3, 4, 0]	[1]
1	[3, 4, 0, 1]	[8]
8	[3, 4, 0, 1, 8]	[1]
1 (déjà vu)	[3, 4, 0, 1, 8]	[]

5 SQL

14. On considère la requête suivante :

```
SELECT e.prenom, e.nom, AVG(n.note) AS m
FROM eleves AS e JOIN notes AS n
    ON e.id=n.eleve
WHERE n.matiere = "Mathématiques"
GROUP BY e.id
HAVING m > 10
ORDER BY e.nom ASC, e.prenom DESC
LIMIT 5
```

- Expliquer simplement en français ce que renvoie cette requête
- En regardant la requête précédente : quelle peut être le schéma relationnel de cette base de données ?

- Cette requête renvoie les prénom/nom/moyenne des élèves qui ont plus de 10 de moyenne en Mathématiques. Le résultat est limité à 5 élèves après avoir trié par nom croissant (puis prénom décroissant en cas d'égalité)
- Schéma relationnel (peut-être partiel) : on ne connait pas le type des identifiants ; pour la clé primaire on peut souligner si on préfère.
  `eleves (id INTEGER PRIMARY KEY, prenom TEXT, nom TEXT)`
  `notes  (eleve INTEGER, matiere TEXT, note REAL)`

15. Expliquer chaque mot-clé qui intervient dans cette requête (utiliser le vocabulaire de l'algèbre relationnelle : sélection, projection, ...).

- Mots-clés :
  - FROM ... JOIN ... ON permet de faire une jointure entre les deux tables eleves et notes (renommées `e` et `n`)
  - WHERE ... : sélection (sur les lignes)
  - GROUP BY ... / AVG(note) : calcul d'agrégat (moyenne de chaque élève)
  - HAVING ... : encore une sélection, mais sur le résultat de l'agrégat
  - SELECT ... : projection
  - ORDER BY / LIMIT : pas vraiment d'équivalent en algèbre relationnelle