TP 14 : Révisions

Exercice 1.

(1) On suppose que des mesures sont réalisées à intervalles réguliers à la fréquence 2 kHz durant 7 secondes. Chaque mesure étant enregistrée comme un flottant codé sur 32 bits, donner en octets l'espace mémoire occupé pour enregistrer toutes ces mesures.

(2) Écrire une fonction **`moyenne_derniers(L,k)`** qui calcule et renvoie la moyenne des k derniers éléments de la liste **`L`**. Pour rédiger cette fonction, on fera l'hypothèse que la liste L donnée en paramètre est bien une liste de nombres contenant au moins k éléments.

On considère l'équation différentielle avec condition initiale :

$$\tau\frac{\mathrm{d}s(t)}{\mathrm{d}t}+s(t)=C$$
$$s(t=0)=0$$

On définit ci-dessous quelques valeurs numériques puis on construit la liste **`t`** $=[t_0,\dots,t_{N-1}]$ avec $t_i=i\times\Delta_t$ pour $i\in[\![0,N-1]\!]$:

```
N = 200
Delta_t = 10
tau = 70
C = 20
```

```
t = []
for i in range(0,N):
    t.append(i*Delta_t)
```

On souhaite déterminer une approximation de la solution de l'équation différentielle aux instants t_i. On note s_i la valeur approchée de s à l'instant t_i et souhaite donc construire la liste **`s`**$=[s_0,\dots,s_{N-1}]$. On approche la valeur de la dérivée par un taux d'accroissement :

$$\frac{\mathrm{d}s}{\mathrm{d}t}(t=t_i)\simeq\frac{s(t_{i+1})-s(t_i)}{t_{i+1}-t_i}\simeq\frac{s[i+1]-s[i]}{\Delta_t}$$

(c) Donner la formulation approchée de l'équation différentielle à l'instant t_i.

(d) Écrire l'expression de $s[i+1]$ en fonction de $s[i]$ obtenue à l'aide de cette formulation approchée. Pour quels indices i cette relation est-elle valable ?

(e) Écrire le code permettant de construire la liste **`s`**.

Exercice 2 *(Réseaux sociaux, d'après Polytechnique, PC, 2016).* On utilise dans cet exercice des listes pour modéliser les relations entre les différents usagers d'un réseau social. Pour créer et manipuler des listes, on utilisera *uniquement* les constructions suivantes :

- **`[]`** crée une nouvelle liste vide et **`[a]*n`** crée une nouvelle liste de taille n dont tous les éléments sont égaux à a ;
- Si **`L`** est une liste de taille n et $i\in[\![0,n-1]\!]$, alors $L[i]$ permet d'accéder à l'élément d'indice i de la liste **`L`**, **`L.append(x)`** modifie la liste **`L`** en ajoutant l'élément **`x`** à la fin et **`L.pop()`** retire le dernier élément de la liste et renvoie sa valeur.

Nous supposerons que les individus sont numérotés de 0 à $n-1$ où n est le nombre total d'individus. Nous représenterons chaque lien d'amitié entre deux individus i et j par une liste contenant leurs deux numéros dans un ordre quelconque, c.-à-d. par la liste $[i,j]$ ou par la liste $[j,i]$ indifféremment. Un réseau social R entre n individus sera représenté par une liste **`reseau`** à deux éléments où :

- **`reseau[0]`** = n contient le nombre d'individus appartenant au réseau ;
- **`reseau[1]`** = la liste *non-ordonnée* (et potentiellement vide) des liens d'amitié déclarés entre les individus.

La figure 1 donne l'exemple d'un réseau social et d'une représentation possible sous la forme de liste. Chaque lien d'amitié entre deux personnes est représenté par un trait entre elles.

```
reseau = [8, [[0,1],[1,3],[3,2],[2,0],[0,3],[2,1],[4,5],
              [5,7],[7,6],[6,4],[7,4],[6,5],[2,4],[5,3]] ]
```

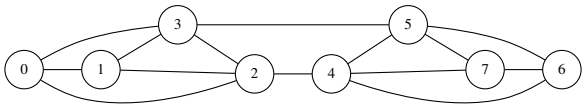
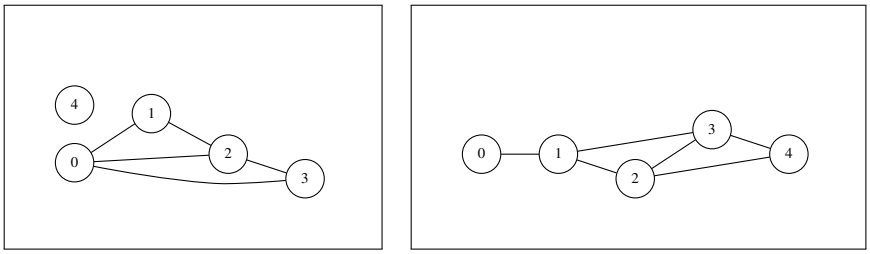


Figure 1 – Un réseau à 8 individus ayant 14 liens d'amitié déclarés et une de ses représentations possibles en mémoire sous forme d'une liste [Nombre d'individus, liste des liens d'amitié].

(1) Donner une représentation sous forme de listes pour chacun des deux réseaux sociaux ci-dessous :



(2) Écrire une fonction **`creerReseauVide(n)`** qui crée, initialise et renvoie la représentation sous forme de liste du réseau à n individus n'ayant aucun lien d'amitié déclaré entre eux.

(3) Écrire une fonction **`estUnLienEntre(paire,i,j)`** où **`paire`** est une liste à deux éléments et i et j sont deux entiers, et qui renvoie **`True`** si les deux éléments contenus dans **`paire`** sont **`i`** et **`j`** dans un ordre quelconque ; et renvoie **`False`** sinon.

(4) Écrire une fonction **`sontAmis(reseau,i,j)`** qui renvoie **`True`** s'il existe un lien d'amitié entre les individus **`i`** et **`j`** dans le réseau **`reseau`** ; et renvoie **`False`** sinon. Quelle est la complexité en temps de votre fonction dans le pire cas en fonction du nombre m de liens d'amitié déclarés dans le réseau ?

(5) Écrire une procédure **`declareAmis(reseau,i,j)`** qui modifie le réseau **`reseau`** pour y ajouter le lien d'amitié entre les individus **`i`** et **`j`** *si ce lien n'y figure pas déjà.* Quelle est la complexité en temps de votre procédure dans le pire cas en fonction du nombre m de liens d'amitié déclarés dans le réseau ?

(6) Écrire une fonction **`listeDesAmisDe(reseau,i)`** qui renvoie la liste des amis de **`i`** dans le réseau **`reseau`**. Quelle est la complexité en temps de votre fonction dans le pire cas en fonction du nombre m de liens d'amitié déclarés dans le réseau ?

Corrections

Ex 1. (1) On réalise des mesures à la fréquence 2 kHz, il y a donc 2000 mesures par secondes. Sur 7 secondes cela donne 14000 mesures et chaque mesure occupe 32 bits soit 4 octets. L'espace mémoire occupé est donc de 56000 octets.

(2) Notons n la taille de la liste. Il suffit de calculer la somme, notée s, des k derniers termes de la liste puis diviser par k. Ces termes sont situés aux indices $n-k, n-k+1, \ldots, n-1$ et on peut donc utiliser une boucle *for* avec la construction **`range(n-k,n)`** afin d'ajouter chaque terme successivement à une variable **`s`** initialement nulle.

```
def moyenne_derniers(L,k):
    n = len(L)
    s = 0
    for i in range(n-k,n):
        # n-k<=i<=n-1
        s = s+L[i]
    return s/k
```

On peut également écrire :

```
def moyenne_derniers(L,k):
    n = len(L)
    for i in range(1,k+1):
        # 1<=i<=k
        s = s+L[n-i]
    return s/k
```

(3) À l'instant t_i l'équation différentielle s'écrit :

$$\tau \frac{\mathrm{d}s}{\mathrm{d}t}(t_i) + s(t_i) = C$$

En utilisant les approximations $s(t_i) \simeq s[i]$ et $\dfrac{\mathrm{d}s}{\mathrm{d}t}(t_i) \simeq \dfrac{s[i+1]-s[i]}{\Delta_t}$ on obtient la formulation approchée :

$$\tau \frac{s[i+1]-s[i]}{\Delta_t} + s[i] = C$$

(4) On isole le terme $s[i+1]$:

$$s[i+1] = s[i] + \frac{\Delta_t}{\tau}(C - s[i])$$

Les indices de la liste s sont compris au sens large entre 0 et $N-1$, on doit donc avoir $0 \leqslant i+1 \leqslant N-1$ et $0 \leqslant i \leqslant N-1$ soit $0 \leqslant i \leqslant N-2$.

(5) On utilise une liste **`s`** contenant initialement la valeur 0 (condition initiale) puis on lui ajoute successivement les termes

$$s[i] + \frac{\Delta_t}{\tau}(C - s[i])$$

avec $0 \leqslant i \leqslant N-2$ à l'aide d'une boucle *for*.

```
N = 200
Delta_t = 10
tau = 70
C = 20

t = []
for i in range(0,N):
    t.append(i*Delta_t)

s = [0]
for i in range(0,N-1):
    # 0<=i<=N-2
    s.append(s[i]+Delta_t/tau*(C-s[i]))
```

On peut aussi préférer l'écriture :

```
s = [0]*N
for i in range(0,N-1):
    # 0<=i<=N-2
    s[i+1] = s[i]+Delta_t/tau*(C-s[i])
```

Ex 2. (1) On note **reseau_A** et **reseau_B** ces deux listes :

```
reseau_A = [5,[[0,1],[0,2],[0,3],[1,2],[2,3]]]
reseau_B = [5,[[0,1],[1,2],[1,3],[2,4],[2,3],[3,4]]]
```

(2) Le réseau vide sur n individus est représenté par la liste **[n,[]]** donc :

```
def creerReseauVide(n):
    return [n,[]]
```

(3) Il suffit de vérifier si **paire[0]==i** et **paire[1]==j** ou le contraire :

```
def estUnLienEntre(paire,i,j):
    if (paire[0]==i and paire[1]==j) or (paire[0]==j and paire[1]==i):
        return True
    else:
        return False
```

ou plus rapidement :

```
def estUnLienEntre(paire,i,j):
    return (paire[0]==i and paire[1]==j) or (paire[0]==j and paire[1]==i)
```

(4) On parcourt toutes les paires du réseau et on renvoie **True** dès que l'on en trouve une qui est un lien entre i et j. Dans le cas contraire, on renvoie **False**.

```
def sontAmis(reseau,i,j):
    for paire in reseau[1]:
        if estUnLienEntre(paire,i,j):
            return True
    return False

print(sontAmis(reseau_A,2,4))
print(sontAmis(reseau_B,2,4))
```

```
False True
```

S'il y a m liens d'amitiés, la boucle *for* parcourt (dans le pire cas) une liste de taille m, par ailleurs la fonction **estUnLienEntre** est exécutée en temps constant. Par conséquent le temps d'exécution dans le pire cas de **sontAmis** est $O(m)$.

(5) Pour vérifier si le lien est déjà présent, on utilise la fonction **sontAmis**. L'énoncé demande d'écrire une *procédure* : on ne renvoie donc pas de résultat, c'est la liste **reseau** donnée en paramètre qui est directement modifiée.

```
def declareAmis(reseau,i,j):
    if not sontAmis(reseau,i,j):
        reseau[1].append([i,j])
```

La fonction **append** est réalisée en temps constant, donc la complexité en temps dans le pire cas de **declareAmis** est la même que celle de **sontAmis** c'est à dire $O(m)$.

(6) On va à nouveau parcourir toutes les paires du réseau. Lorsque l'une d'elles contient i, alors on ajoute l'autre élément à une liste L, initialement vide.

```
def listeDesAmis(reseau,i):
    L = []
    for paire in reseau[1]:
        if paire[0]==i:
            L.append(paire[1])
        elif paire[1]==i:
            L.append(paire[0])
    return L
```

On dispose à nouveau d'une boucle *for* qui parcourt les m paires du réseau, le bloc d'instructions de cette boucle est en temps constant et par conséquent la complexité en temps dans le pire cas est $O(m)$.