Comme Mark à Harvard*

INFORMATIQUE COMMUNE - Devoir n° 2 - Olivier Reynet

Consignes :

1. utiliser une copie différente pour chaque partie du sujet,
2. écrire son nom sur chaque copie,
3. écrire de manière lisible et intelligible,
4. préparer une réponse au brouillon avant de la reporter sur la feuille.

L'objectif de cet examen est de programmer les fonctions nécessaires à la création d'un logiciel permettant de gérer un réseau social [1]. Les individus du réseau social sont numérotés de 0 à $n-1$ où n est le nombre total d'individus. On dit que n est la taille du réseau. Chaque individu possède un certain nombre de liens d'amitié qui sont implémentés par une liste en Python. Si j est ami avec i alors il existe une liste d'entiers des amis de i notée `Ai` qui contient au moins j et une liste d'entiers `Aj` qui contient au moins i.

Un réseau social R entre n individus sera représenté par une liste de liste d'amitiés `R` : la liste d'amitiés de l'individu de numéro i est stockée en $R[i]$.

Sur la figure 1, on a représenté le réseau social définie par la liste :

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

Les amis de 0 sont `[1, 2, 3]` et les amis de 6 sont `[4, 5, 7]`.

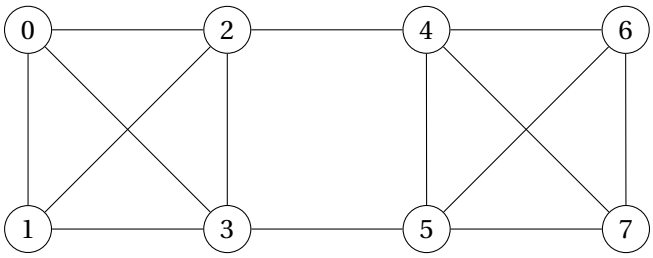

FIGURE 1 – Réseau social à 8 individus. Dans chaque cercle figure un individu, chaque trait représente un lien d'amitié

*ou presque...

1. comme Mark Zuckerberg dans sa chambre d'étudiant à Harvard en 2003

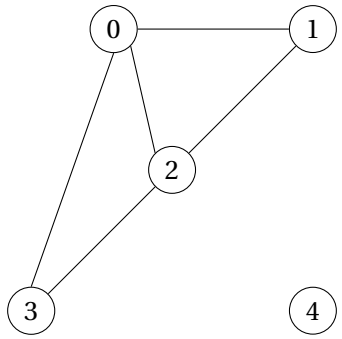

FIGURE 2 – Réseau social R_a

A Partie A : préliminaires

A1. Sur le réseau social de la figure 2, donner :

(a) La liste `A2` des amis de 2.

Solution :

```
A2  = [0,1,3]
```

(b) La liste `A4` des amis de 4.

Solution :

```
A4 = []
```

A2. Un individu représenté par l'entier p est ami avec tous les individus dont le numéro est strictement inférieur à p. Créer la liste `Ap` qui représente les amis de p.

Solution :

```
Ap = []
for i in range(p):
    Ap.append(i)
```

A3. Un ami de quartier est ami avec tous les individus de numéro compris dans l'intervalle entier $[\![i, j]\!]$. Écrire une fonction de signature `amis_de_quartier(i: int, j: int)-> list[int]` qui renvoie la liste des amis d'un individu de quartier $[\![i, j]\!]$.

Solution :

```
def amis_de_quartier(i: int, j: int) -> list[int]:
    Q = []
    for k in range(i, j + 1):
```

```
        Q.append(k)
    return Q
```

A4. Une star est amie avec tous les autres membres d'un réseau. Écrire une fonction de signature `star(s: int, n: int)—> list[int]` qui renvoie la liste de tous les amis d'une star de numéro `s` pour un réseau comportant `n` individus. Attention à ne pas créer un lien entre `s` et `s`.

Solution :

```
def star(s: int, n: int) —> list[int]:
    S = []
    for k in range(n):
        if k != s :
            S.append(k)
    return S
```

A5. On considère la fonction de prototype `est_ami_avec(A, p)` où `A` est une liste d'amis et `p` un individu. Cette fonction statue sur le fait que `p` est présent ou pas dans la liste `A`.

(a) Quel est le type retourné par la fonction `est_ami_avec` ?

Solution : Cette fonction renvoie un booléen `bool`.

(b) Écrire cette fonction en effectuant une recherche séquentielle dans la liste. Il n'est pas autorisé d'utiliser la syntaxe `elem in liste` qui effectue déjà une recherche séquentielle...

Solution :

```
def est_ami_avec(A, p) :
  for k in range(len(A)):
      if p == A[k]:
          return True
  return False
```

A6. On considère la liste d'amis suivante `[2,5,0,9,4,7,1]`. Trier manuellement cette liste à l'aide de l'algorithme de tri par **insertion**. Vous ferez apparaître les étapes nécessaires à la compréhension de la méthode de tri.

Solution : Cf. cours.

```
[2,5,0,9,4,7,1]
[2,5,0,9,4,7,1]
[0,2,5,9,4,7,1]
[0,2,5,9,4,7,1]
[0,2,4,5,9,7,1]
[0,2,4,5,7,9,1]
```

```
[0,1,2,4,5,7,9]
```

A7. Écrire une fonction de prototype `insertion_sort(A)` où `A` est une liste d'amis qui tri `A` en utilisant l'algorithme du tri par insertion.

Solution :

```
def insertion_sort(t):
  for i in range(1, len(t)):
      to_insert = t[i]
      j = i
      while t[j - 1] > to_insert and j > 0:
          t[j] = t[j - 1]
          j -= 1
      t[j] = to_insert
```

B Partie B : mécanique du réseau social

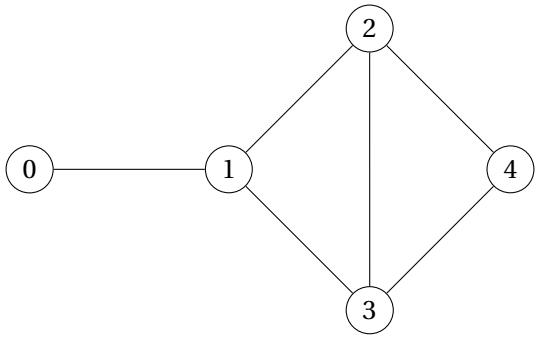

FIGURE 3 – Réseau social R_b

B1. Donner la représentation du réseau social R_b dessiné sur la figure 3 sous la forme d'une liste de listes nommée `Rb`.

Solution :

```
    Rb = [[1],[0,2,3],[1,3,4],[1,2,4],[2,3]]
```

B2. Écrire une fonction de prototype `creer_reseau_vide(n)` qui renvoie un réseau à n individus n'ayant aucun lien d'amitié entre eux.

Solution :

```python
def creer_reseau_vide(n):
    V = []
    for _ in range(n):
        V.append([])
    return V
```

Lors de la déclaration des liens d'amitié, on souhaite insérer les nouveaux numéros dans les listes d'amitié en faisant en sorte que la liste soit triée. Ceci est possible si à chaque fois qu'un élément est inséré, celui-ci est rangé à la bonne place.

B3. Écrire une fonction de prototype `inserer_ami(A, ami)` qui insère l'entier `ami` dans une liste d'amitié `A` à la bonne place, c'est-à-dire dans l'ordre croissant des entiers. Cette fonction travaille en place sur la liste `A`. Par exemple, les instructions `A = [0, 4, 9] ; inserer_ami(A, 7)` transforment la liste `A` en la liste `[0, 4, 7, 9]`.

Solution :

```python
def inserer_ami(A, ami):
    A.append(ami)
    if len(A) > 1 and ami < A[-2]:
        j = len(A) - 1
        while A[j - 1] > ami and j > 0:
            A[j] = A[j - 1]
            j -= 1
        A[j] = ami
```

R **Pour toutes les questions suivantes, les listes d'amitiés d'un réseau sont donc triées de manière ascendante.**

B4. En utilisant l'algorithme de recherche d'un élément par dichotomie, écrire une fonction de prototype `rech_dicho(A, p)-> bool` qui statue sur la présence de `p` dans la liste `A`.

Solution :

```python
def rech_dicho(A, p):
    n = len(A)
    g = 0
    d = n - 1
    while g <= d:
        m = (g + d) // 2
        if A[m] == p:
            return True
        elif A[m] < p:
            g = m + 1
        else:
            d = m - 1
    return False
```

B5. Écrire une fonction de prototype `sont_amis(R, i, j)` qui statue sur le fait que `i` et `j` sont amis dans le réseau `R`. Ces deux individus sont amis si et seulement si `i` est ami et de `j` **et** `j` est ami de `i`. On veillera à coder de manière efficace.

Solution :

```
def sont_amis(R, i, j):
    return rech_dicho(R[i], j) and rech_dicho(R[j], i)
```

B6. Écrire une fonction de prototype `declarer_amis(R, i, j)` qui modifie le réseau `R` afin d'enregistrer le lien d'amitié entre `i` et `j`. On fera attention à ne pas déclarer un individu ami avec lui-même **et** à ne pas déclarer plusieurs fois un lien d'amitié déjà existant. On garantira également que les listes d'amitié restent triées après l'insertion des nouveaux liens.

Solution :

```
def declarer_amis(R, i, j):
    if not sont_amis(R, i, j) and i != j:
        inserer_ami(R[i],j)
        inserer_ami(R[j],i)
```

B7. Écrire une fonction de prototype `generer_reseau_aleatoire(n, na_max)` qui renvoie un réseau de taille `n` généré aléatoirement dans lequel chaque individu possède au plus `na_max` amis. On s'appuiera sur les fonctions précédemment codées. Il est également nécessaire d'utiliser la fonction `randrange` du module `random`. On rappelle que `randrange(n)` génère un entier aléatoirement entre 0 et $n-1$.

Solution :

```
from random import randrange
def generer_reseau_aleatoire(n, na_max):
    R = creer_reseau_vide(n)
    for k in range(n):
        na = randrange(na_max+1)
        for i in range(na):
            a = randrange(n)
            declarer_amis(R, k, a)
    return R
```

C Partie C : mesures et tests

C1. Écrire une fonction de prototype `moins_connecte(R)->` `int` qui renvoie le numéro d'un des individus le moins connecté d'un réseau `R`. Appliquée au réseau R_b, cette fonction renvoie 0.

Solution :

```python
def moins_connecte(R):
    if len(A) > 0:
        m = len(R[0])
        i_max = 0
        for k in range(len(R)):
            if len(R[k]) < m:
                m = len(R[k])
                i_max = k
        return i_max
    else:
        return None
```

C2. Écrire une fonction de prototype `communs_i_j(R, i, j)` qui renvoie la liste des amis communs de `i` et de `j` ou la liste vide s'ils n'en ont pas. **Aucune fonction précédente n'est autorisée.** Profiter du fait que les listes sont triées pour procéder par comparaison successive des termes deux à deux.

Solution :

```python
def communs_i_j(R, i, j):
    k = 0
    p = 0
    communs = []
    while k < len(R[i]) and p < len(R[j]):
        if R[i][k] == R[j][p]:
            communs.append(R[i][k])
            k += 1
            p += 1
        elif R[i][k] < R[j][p]:
            k += 1
        else:
            p += 1
    return communs
```

C3. Écrire une fonction de prototype `rec_communs_i_j(R, i, j, k, p, communs)` qui renvoie la liste des amis communs de `i` et de `j` ou la liste vide s'ils n'en ont pas. Cette fonction procède de manière récursive pour aboutir au même résultat que la fonction précédente. Les indices `k` et `p` permettent de désigner la position des éléments considérés dans les listes `Ai` et `Aj` lors de l'appel récursif. Pour utiliser cette fonction, on procède comme suit : `communs = rec_amis_communs_i_j(RR, 0, 1, 0, 0, [])`.

Solution :

```python
def rec_communs_i_j(R, i, j, k, p, communs):
    if k >= len(R[i]) or p >= len(R[j]):
        return communs
    else:
        if R[i][k] == R[j][p]:
            communs.append(R[i][k])
            k += 1
```

```
            p += 1
        elif R[i][k] < R[j][p]:
            k += 1
        else:
            p += 1
        return rec_communs_i_j(R, i, j, k, p, communs)
```

Le degré de centralité d'un individu i dans un réseau de taille n est définie par :

$$C_i = \frac{|\, A_i \,|}{n-1} \tag{1}$$

où $|\, A_i \,|$ est le nombre d'amis de l'individu i.

C4. Écrire une fonction de prototype `centralite(A, n)—> float` où `A` est la liste d'amis d'un individu et `n` la taille du réseau. Cette fonction renvoie le degré de centralité d'un individu.
Par exemple, `centralite([0, 1, 2], 5)` renvoie `0.75`.

Solution :

```
def centralite(A, n):
    assert n > 1
    return len(A) / (n—1)
```

C5. Écrire une fonction de prototype `plus_central(R)` où `R` est un réseau. Cette fonction renvoie le numéro de l'individu dont le degré de centralité est le plus grand ainsi que sa centralité dans le réseau, le tout sous la forme d'un tuple. **On veillera à n'utiliser qu'une seule boucle. La fonction max n'est pas autorisée.** Par exemple, `plus_central([[3, 4], [3], [3], [0, 1, 2], [0]])` renvoie `(3, 0.75)`.

Solution :

```
def plus_central(R):
    n = len(R)
    assert n > 0
    max_c = 0.0  # convient car la centralité est comprise entre 0 et 1
    imax_c = 0  # indice du plus central
    for i in range(len(R)):
        cc = centralite(R[i], n)  # centralité calculée pour un individu i
        if cc > max_c:
            imax_c = i
            max_c = cc
    return imax_c, max_c
```

C6. Écrire une fonction de prototype `k_centres(R, k)` où `R` est un réseau et `k` un entier strictement positif. Cette fonction renvoie les numéros des `k` individus dont le degré de centralité est le plus grand dans le réseau. On pourra s'aider de l'algorithme du tri par insertion pour trier une liste conformément à une autre, simultanément. **La fonction `centralite` est la seule fonction précédente autorisée.**

Solution :

```
1  def k_centres(R, k):
2      assert k > 0
3      n = len(R)
4      C = []
5      individus = [i for i in range(n)]
6      for i in range(len(R)):
7          cc = centralite(R[i], n)
8          C.append(cc)
9      for i in range(1, len(C)):  # tri des individus dans l'ordre décroissant de centralité
10         to_insert = C[i]
11         gus_to_move = i
12         j = i
13         while C[j - 1] < to_insert and j > 0:
14             C[j] = C[j - 1]
15             individus[j] = individus[j - 1]
16             j -= 1
17         C[j] = to_insert
18         individus[j] = gus_to_move
19     return individus[:k]
```

On dispose de la fonction `mystere` suivante :

```
1  def mystere(i, R):
2      stack = [i]
3      seen = [False for _ in range(len(R))]
4      world = []
5      while len(stack) > 0:
6          a = stack.pop()
7          if not seen[a]:
8              world.append(a)
9              seen[a] = True
10             for friend in R[a]:
11                 stack.append(friend)
12     return world
```

C7. On cherche à comprendre comment fonctionne la fonction `mystere`. Pour cela, on s'appuie sur les numéros de ligne dans le code.

(a) Quel est le résultat des instructions des lignes 10 et 11 ?

Solution : Tous les amis de `a` sont ajoutés à la liste `stack`.

(b) Que fait l'instruction `stack.pop()` à la ligne 6 ?

Solution : Elle renvoie le dernier élément de la liste `stack` et le supprime de la liste. Cet élément est affecté à la variable `a`.

(c) À la ligne 5, donner une interprétation de la condition d'arrêt de la boucle `while`.

Solution : Lorsque tous les amis et les amis des amis ont été visités par l'algorithme, celui-ci se termine. La liste `seen` permet de savoir si on a déjà rencontré un ami.

(d) Que renvoie la fonction `mystere` ?

Solution : Elle renvoie la liste des individus que i peut espérer atteindre grâce à son réseau d'amis.

C8. En vous inspirant de la fonction `mystere`, écrire une fonction `sont_joignables(i, j, R)` qui permet de statuer qu'un individu i peut joindre j dans un réseau R.

Solution :

```
def sont_joignables(i, j, R):
    a_voir = [i]
    deja_vus = [False for _ in range(len(R))]
    while len(a_voir) > 0:
        a = a_voir.pop()
        if a == j:
            return True
        if not deja_vus[a]:
            deja_vus[a] = True
            for ami in R[a]:
                a_voir.append(ami)
    return False

# USAGE
R = generer_reseau_aleatoire(42, 5)
print(sont_joignables(0, 1, R))
```