Informatique, concours blanc. Une correction

A- Programmation en Python

Partie I. Réseaux sociaux

1. `Reseau_A = [5, [[0,1], [0,2], [0,3], [1,2], [2,3]]]`

   `Reseau_B = [5, [[0,1], [1,2], [1,3], [2,3], [2,4], [3,4]]]`

2. Une solution possible :

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

3. Une réponse possible :

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

4. • La fonction demandée peut-être :

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

   • Dans le pire des cas, i et j ne sont pas amis et la complexité est en $O(m)$ (m tours de boucle à temps constant).

5. • La procédure demandée peut-être :

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

   • Dans le pire des cas, la complexité de `sontAmis(reseau, i, j)` est en $O(m)$ donc c'est aussi le cas de `declareAmis(reseau, i, j)`.

6. • La fonction demandée peut-être :

```
def listeDesAmisDe(reseau, i):
    amis_i = []
    for j, k in reseau[1]:
        if j == i:
            amis_i.append(k)
        elif k == i:
            amis_i.append(j)
    return amis_i
```

   • Dans le pire des cas, tous les liens vont relier i (en deuxième position) à un autre individu, ce qui donne un nombre constant d'opérations à chacun des m tours de boucle : la complexité est en $O(m)$.

Partie II. Partitions

7. • Représentation filiale A
   Représentants : 5, 4, 1 et 3.

i :	0	1	2	3	4	5	6	7	8	9
`parent`$[i]$:	5	1	1	3	4	5	1	5	5	7

   • Représentation filiale B
   Représentants : 9, 7 et 3.

i :	0	1	2	3	4	5	6	7	8	9
`parent`$[i]$:	3	9	0	3	9	4	4	7	1	9

8. La fonction demandée peut-être :

```
def creerPartitionEnSingletons(n):
    parent = []
    for i in range(n):
        parent.append(i)
    return parent
```

9. • Une version récursive de la fonction demandée est :

```
def representant(parent, i):
    if parent[i] == i:
        return i
    else:
        return representant(parent, parent[i])
```

• Une version Version itérative de la fonction est :

```
def representant(parent, i):
    ancetre = i
    son_pere = parent[ancetre]
    while son_pere != ancetre:
        ancetre = son_pere
        son_pere = parent[ancetre]
    return ancetre
```

• Dans le pire des cas, on a un seul groupe : i est au pied (ou plutôt feuille) de cet arbre à une branche. La complexité est donc en $O(n)$.

• Exemple : $i = 0$ et

i :	0	1	2	3	4	5	6	...	$n-2$	$n-1$
parent$[i]$:	1	2	3	4	5	6	7	...	$n-1$	$n-1$

10. La procédure demandée peut-être :

```
def fusion(parent, i, j):
    p = representant(parent, i)
    q = representant(parent, j)
    parent[p] = q
```

11. On exécute dans l'ordre :

1. `fusion(parent,0,1)`
2. `fusion(parent,0,2)`

⋮

(n-1). `fusion(parent,0,n-1)`

Le groupe de 0 a comme tailles successives $1, 2, \ldots, n-1$ et 0 s'y trouve toujours en bas (feuille de l'arbre) ; ainsi les calculs de son représentant vont avoir un coût de l'ordre de $1+2+\cdots+(n-1) = \dfrac{n(n-1)}{2}$. Ce qui reste est négligeable.

On aura bien une fusion totale en n^2 opérations élémentaires.

12. • Une version itérative peut-être :

```
def representant(parent, i):
    liste_ancetres = [i]
    ancetre = i
    son_pere = parent[ancetre]
    while son_pere != ancetre:
        liste_ancetres.append(son_pere)
        ancetre = son_pere
        son_pere = parent[ancetre]
        representant = ancetre
    for ancetre in liste_ancetres:
        parent[ancetre] = representant
    return ancetre
```

• Une version récursive peut-être :

```
def representant(parent, i):
    if parent[i] == i:
        return i
    else:
        rep = representent(parent,parent[i])
        parent[i] = rep
        return rep
```

• Comme cela rajoute un coût au plus linéaire à un coût qui l'est déjà, la complexité est toujours en $O(n)$.

13. La fonction demandée peut-être :

```
def listeDesGroupes(parent):
    n = len(parent)
    position_representant = [-1] * n
            # position_representant[i] contient soit -1, lorsque i nést pas un représentant c
            # connu, soit la position à laquelle il a été rencontré sinon.
    liste_groupes = []
    for i in range(n):
        rep = representant(parent, i)
        pos = position_representant[rep]
        if pos == -1:
                # représentant jamais rencontré : nouveau groupe
            position_representant[rep] = len(liste_groupes)
            liste_groupes.append([i])
        else:
            liste_groupes[pos].append(i)
                # repr. déjà rencontré, ajout au groupe corresp.
    return liste_groupes
```

Partie III. Algorithme randomisé pour la coupe minimum

14. Une proposition est :

```
def echange(L, i, j):
    L[i], L[j] = L[j], L[i]

def pasMemeGroupe(parent, i, j):
        return representant(parent, i) != representant(parent, j)
                #True si i et j ne sont pas dans le même groupe, False sinon.

def coupeMinimumRandomisee(reseau):
    n, liens = reseau
    m = len(liens)    # Nombre de liens
    nombre_groupes = n
    P = creerPartitionEnSingletons(n)
    indice_max_non_marque = m - 1
    while nombre_groupes > 2 and indice_max_non_marque >= 0:
        k = random.randint(0,indice_max_non_marque)
        i, j = liens[k]
        if pasMemeGroupe(P, i, j):
            fusion(P, i, j)
            nombre_groupes =nombre_groupes-1
        echange(liens, k, indices_non_marques)
        indice_max_non_marque=indice_max_non_marque-1
#On choisit de fusionner le groupe de 0 avec k - 2 autres groupes.
    i = 1
    while nombre_groupes > 2:
        if pasMemeGroupe(P, i, 0):
            fusion(P, i, 0)
            nombre_groupes= nombre_groupes-1
        i=i+1
    return P
```

• **Complexité**

— Pour passer de n à 2 groupes, il faut exactement $n-2$ fusions, chacune en $2\alpha(n)$, soit $O(n\alpha(n))$.

— Il y a $k+\ell$ tests de `pasMemeGroupe` où k le le nombre de tours dans la première boucle et ℓ le nombre de tours dans le deuxième, avec $k \leqslant m$ (et même $\min(n,m)$) et $\ell \leqslant n$ soit $O((n+m)\alpha(n))$.
— Le reste est négligeable.
— On obtient au total une complexité en $O((n+m)\alpha(n))$.

15. Une fonction qui pourrait convenir est :

```
def tailleCoupe(reseau, parent):
    _, liens = reseau
    nombre_liens = 0
        for i, j in liens:
            if pasMemeGroupe(parent, i, j):
                nombre_liens = nombre_liens+1
    return nombre_liens
```

B - Programmation SQL

16.
```
SELECT
    id2
FROM
    LIENS
WHERE id1 = x;
```

17.
```
SELECT
    nom,
    prenom
FROM
    INDIVIDUS
JOIN
    LIENS
        ON id = id1
WHERE id2 = x;
```

18.
```
SELECT
    L1.id1
FROM
    LIENS AS L1
JOIN
    LIENS AS L2
        ON
            L2.id1 = L1.id2
WHERE L2.id2 = x;
```

ou bien

```
SELECT
    id1
FROM
    LIENS
WHERE id2 IN (SELECT id1 FROM LIENS
                WHERE id2 = x);
```

ou bien

```
SELECT
    id1
FROM
    LIEN AS L
WHERE EXISTS (SELECT * FROM LIENS
                WHERE id2 = x AND id1 = L.id2);
```