CORRIGÉ DS N°2 INFO : X PC 2016

1.

```
# Question 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.

```
# Question 2

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

3.

```
# Question 3

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

4.

```
# Question 4

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

La fonction `estUnLienEntre` est de complexité O(1) (coût constant).
Dans la fonction `sontAmis` , dans le pire des cas, c'est-à-dire si `i` et `j` ne sont pas amis, la liste `reseau[1]` est parcourue en entier.
La complexité est donc en $\mathcal{O}(m)$, où m désigne le nombre de liens d'amitié déclarés.

5.

```
# Question 5

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

La méthode `append` est de complexité $\mathcal{O}(1)$ donc la complexité de la fonction `declareAmis` est celle de la fonction `sontAmis`, soit $\mathcal{O}(m)$.

6.

```
# Question 6

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

```
                amis_De_i.append(paire[0])
        return amis_De_i
```

La complexité de la fonction `listeDesAmisDe` est proportionnelle à la longueur de la liste `reseau[1]`, soit $O(m)$.

7.

```
# Question 7

parentA = [5, 1, 1, 3, 4, 5, 1, 5, 5, 7]
# Les représentants des 4 groupes sont 5, 4, 1 et 3 .

parentB = [3, 9, 0, 3, 9, 4, 4, 7, 1, 9]
# Les représentants des 3 groupes sont 9,7 et 3 .
```

8.

```
# Question 8

# 1ère solution:
def creerPartitionEnSingletons(n):
    parent = []
    for i in range(n):
       parent.append(i)
    return parent

# 2ème solution
def creerPartitionEnSingletons(n):
    parent = [0]*n
    for i in range(n):
       parent[i] = i
    return parent

# 3ème solution, la plus élégante
def creerPartitionEnSingletons(n):
    return [i for i in range(n)]
```

9.

```
# Question 9

# 1ère solution
def representant(parent,i):
    fils = i
    pere = parent[fils]
    while pere != fils:
        fils = pere
        pere  = parent[fils]
    return pere

# 2ème solution, plus concise
def representant(parent, i):
    while parent[i] != i:
        i = parent[i]
    return i
```

Le pire des cas pour cette fonction correspond au cas où l'on n'a qu'un seul groupe et que l'on cherche le représentant de l'élément le plus loin dans la chaîne, par exemple lorsque

`parent = [ 1, 2, 3, 4, ....., n-2, n-1, n-1]`

et que l'on cherche le représentant de 0.
Dans ce cas, il faut parcourir toute la liste, et la complexité est en $O(n)$.

10.

```
# Question 10

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

11. On considère la suite de fusions suivante :

```
fusion(parent, 0, 1)
fusion(parent, 0, 2)
fusion(parent, 0, 3)
...
fusion(parent, 0, n-1)
```

Le groupe de 0 a pour tailles successives $1,\ 2,\ \ldots,\ n-1$ et à chaque fois il faut le parcourir en entier pour trouver le représentant de 0. Il y aura donc $1+2+...+(n-1)=\dfrac{n(n-1)}{2}$ opérations, soit une complexité $O(n^2)$.

12.

```
# Question 12

# Version itérative
def representant (parent, i):
    Liste_ancetres = [i]
    # liste des ancetres de i
    fils = i
    pere = parent [fils]
    while pere != fils :
        Liste_ancetres.append (pere)
        fils = pere
        pere = parent [fils]

    for element in Liste_ancetres :
        parent [ element ] = pere
    return pere

# Version récursive
def representant (parent, i):
    if parent [i] == i:
        return i
    else :
        j = representant (parent, parent [i])
        parent [i] = j
        return j
```

On voit sur la version itérative que cette fonction ajoute le parcours de `Listeancetres` par rapport à la version initiale. Le coût de ce parcours est au plus en $O(n)$, la complexité finale est donc $O(n)+O(n)=O(n)$, inchangée.

13. La fonction qui suit utilise deux listes : `groupes` qui contient les groupes en voie de formation et `rep` qui contient les représentants des groupes déjà rencontrés. Pour chaque élément `i` de la liste parent, on calcule le représentant `r` de son groupe. S'il n'est pas déjà présent dans la liste `rep` , il y est ajouté et un nouveau groupe vide est créé dans la liste groupes. On ajoute ensuite `i` au groupe associé à `r` .

Comme indiqué dans l'énoncé, on n'utilise pas de fonction sophistiquée sur les listes, ce qui alourdit un peu la programmation.

```
# Question 13

def listeDesGroupes(parent):
```

```
    groupes = []
    rep = []
    for i in range(len(parent)):
        r = representant(parent, i)
        k = 0
        while k < len(rep) and rep[k] != r:
            k += 1
        if k == len(rep):
            rep.append(r)
            groupes.append([])
        groupes[k].append(i)
    return groupes
```

14. La fonction se contente de suivre pas-à-pas l'algorithme décrit dans l'énoncé.
La variable `nbLiensNonMarques` contient le nombre de liens non marqués ; les liens non marqués sont donc les `nbLiensNonMarques` premiers dans la liste des liens du réseau, les liens marqués sont les derniers.

```
# Question 14

def coupeMinimumRandomisee(reseau):
    # Etapes 1 et 2 de l'algorithme: initialisations
    n = reseau[0]
    nbGroupes = n
    nbLiensNonMarques= len(reseau[1])
    parent = creerPartitionEnSingletons(n)

    # Etape 3 de l'algorithme
    while nbGroupes > 2 and nbLiensNonMarques > 0:
        # (a)
        k = random.randint(0, nbLiensNonMarques)
        [i, j] = reseau[1][k]
        # (b)
        ri = representant(parent, i)
        rj = representant(parent, j)
        if ri != rj:
            fusion(parent, ri, rj)
            nbGroupes -= 1
        # (c)
        reseau[1][k], reseau[1][nbLiens] = reseau[1][nbLiens], reseau[1][k]
        nbLiensNonMarques -= 1

    # Etape 4 : on choisit de fusionner le groupe de 0 avec k-1 autres groupes
    if nbGroupes > 2:
        r0 = representant(parent, 0)
        i = 1
        while nbGroupes > 2:
            ri = representant(parent, i)
            if r0 != ri:
                fusion(parent, r0, ri)
                nbGroupes -= 1
            i += 1

    # Etape 5
    return parent
```

Pour passer d'une partition en n groupes à une partition en 2 groupes il faut réaliser $n-2$ fusions ; la complexité d'une fusion étant en $O(\alpha(n))$, le coût total de ces fusions est en $O(n\alpha(n))$.

Dans le pire des cas, pour chaque lien présent on détermine le représentant de la classe de chacun des deux protagonistes, ce qui occasionne un coût en $O(m\alpha(n))$.

Enfin, l'étape 4 peut dans le pire des cas occasionner la recherche de $n-1$ représentants dans le tableau `parent` , avec là encore un coût total en $O(n\alpha(n))$.

En définitive, la complexité totale de cette fonction est un $\mathcal{O}((n+m)\alpha(n))$.

15.

```python
# Question 15

def tailleCoupe(reseau, parent):
    nbLiens = 0
    for [i, j] in reseau[1]:
        if representant(parent, i) != representant(parent, j):
        nbLiens += 1
    return nbLiens
```

16.

```
SELECT id1 FROM LIENS WHERE id2 = x
```

17.

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

ou bien :

```
SELECT nom , prenom FROM INDIVIDUS , LIENS WHERE id2 = x AND id = id1
```

ou bien :

```
SELECT nom , prenom  FROM INDIVIDUS WHERE id IN (
SELECT id1 FROM LIENS WHERE id2 = x
)
```

18.

```
SELECT b.id2 FROM LIENS a JOIN LIENS b ON a.id2 = b.id1 WHERE a.id1 = x
```

ou bien :

```
SELECT b.id2 FROM LIENS AS a, LIENS AS b WHERE a.id2 = b.id1 AND a.id1 = x
```

ou bien :

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