# Épreuve d'informatique de l'X - 2016 - MP/PC

## Partie I. Réseaux sociaux

**Question 1** : Voici une représentation sous forme de listes des réseaux $A$ et $B$ :

```
ReseauA = [5,[[0,1],[0,2],[0,3],[2,1],[2,3]]]
ReseauB = [5,[[0,1],[1,2],[1,3],[2,3],[2,4],[3,4]]]
```

**Question 2** : fonction `creerReseauVide` de création d'un réseau de $n$ individus sans lien d'amitié.

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

**Question 3** : Fonction `estUnLienEntre` qui teste si la paire `paire` est égale à la paire $(i, j)$ (ou $(j, i)$).

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

**Question 4** : On parcourt la liste des liens jusqu'à ce que l'on trouve un lien entre $i$ et $j$ ou d'avoir parcouru toute la liste des liens.

```
def sontAmis(reseau,i,j):
    liste = reseau[1]
    m = len(liste)
    trouve = False
    k = 0
    while not(trouve) and k < m:
        trouve = estUnLienEntre(liste[k],i,j)
        k += 1
    return(trouve)
```

Dans le pire des cas, les individus $i$ et $j$ n'ont pas de liens d'amitié, il faut donc parcourir entièrement la liste des liens d'amitié. La complexité de la fonction `sontAmis` dans le pire des cas est en $O(m)$ où $m$ est le nombre de liens d'amitié.



(PSI*)

**Question 5** : Fonction `declareAmis` qui ajoute un lien entre $i$ et $j$ dans le réseau `reseau` si celui n'est pas déjà présent.

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

Les opérations de mise à jour de la variable `reseau` sont à coût constant, mais il faut prendre en compte le coût de l'appel à la fonction `sontAmis`.

La complexité de la fonction `declareAmis` dans le pire des cas est égal à la complexité de la fonction `sontAmis` donc elle est en $O(m)$ où $m$ est le nombre de liens d'amitié.

**Question 6** : La fonction `listeDesAmisDe(reseau, i)` renvoie la liste des amis de $i$ dans le réseau `reseau`. On parcourt une seule fois la liste des liens d'amitié.

```
def listeDesAmisDe(reseau, i):
    amis = []
    liste = reseau[1]
    m = len(liste)
    for k in range(m):
        k,l = liste[k]
        if k == i:
            amis.append(l)
        if l == i:
            amis.append(k)
    return(amis)
```

La complexité dans le pire des cas est en $O(m)$ où $m$ est le nombre de liens d'amitié.

## Partie II. Partitions

**Question 7** : Voici les tableaux `parentA` et `parentB` encodant les représentations filiales des partitions $A$ et $B$ de l'énoncé :

```
parentA = [5,1,1,3,4,5,1,5,5,7]
parentB = [3,9,0,3,9,4,4,7,1,9]
```

**Question 8** : La fonction `creerPartitionEnSingletons` crée et renvoie un tableau `parent` à $n$ éléments vérifiant $\forall i \in [\![n]\!]$, `parent`$[i] = i$ :

```
def creerPartitionEnSingletons(n):
    parent = [0]*n
    for i in range(n):
        parent[i] = i
    return(parent)
```

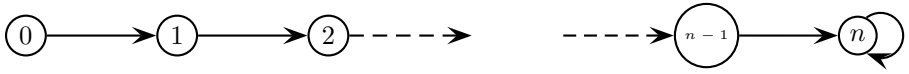
**Question 9** : Un représentant $i$ d'un groupe est caractérisé par le fait que `parent[i]==i`. Le principe est de remonter à partir d'un individu quelconque de parent en parent jusqu'au représentant du groupe. Cela se traduit naturellement par la fonction récursive `representant` suivante :

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

Le nombre d'appels à la fonction `representant` est au plus égal à $n$. Il est égal à $n$ pour l'appel `representant(parent,0)` avec le tableau `parent` défini par

$$\texttt{parent}[i] = \begin{cases} i+1 & \text{si } i \in [\![n-2]\!] \\ n-1 & \text{si } i = n-1 \end{cases}.$$

Dont on peut donner la représentation graphique suivante :




**Question 10** : On écrit la fonction `fusion` en appliquant l'algorithme décrit par l'énoncé :

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

**Question 11** : On part du tableau `parent` tel que $\forall i \in [\![n]\!]$, `parent`$[i] = i$. Pour $i$ variant de 1 à $n-1$ on effectue la fusion `fusion(0,i)`. À la $i$-ème étape, parent$[j] = j+1$ pour $j \in [\![i-1]\!]$ et parent$[j] = j$ pour $j \geqslant i$. À l'appel de `fusion(parent,0,i+1)`, la recherche du représentant de 0 à un coût en $O(i)$ (on «remonte» de la position 0 à la position $i$) et on déclare que le parent de $i$ est $i+1$. La suite des $(n-1)$ fusions nécessite $\dfrac{n(n-1)}{2}$ opérations élémentaires, soit un coût en $O(n^2)$.

**Question 12** : On modifie la fonction `representant` pour modifier le tableau `parent` à chaque appel de la fonction `representant`. Ainsi, pour tout individu $i$ visité, `parent`$[i]$ est actualisé et prend la valeur du représentant du groupe.

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

On peut considérer cette optimisation comme gratuite du point de vue de la complexité car on profite de l'appel à la fonction `representant` pour optimiser le tableau `parent` en ajoutant une seule opération d'affectation.

**Question 13** : On part d'une liste vide nommée `groupes`. En sortie de la fonction `listeDesGroupes`, `groupes` est une liste de listes. Chaque liste représente un groupe et le premier élément de la liste est le représentant du groupe.
On étudie chaque individu par la boucle des lignes 3 à 18. On récupère le représentant $p$ du groupe de $i$ à la ligne 4. Dans la boucle conditionnelle des lignes 8 à 15, on regarde si l'un des groupes en cours de constitution admet $p$ comme représentant. Si c'est le cas et que $i$ n'est pas le représentant du groupe, on ajoute $i$ au groupe.
Si on n'a pas réussi à placer l'individu dans l'un des groupes en cours de constitution, on crée un nouveau groupe (lignes 16 à 18).

```
def listeDesGroupes(parent):
    n = len(parent); groupes = []
    for i in range(n):
        p = representant(parent,i)
        place = False
        m = len(groupes)
        k = 0
        while not(place) and k < m:
            q = groupes[k][0]  # on récupère le représentant du groupe
            if q == p and i != p:
                groupes[k].append(i)
                place = True
            if q == p:
                place == True
            k +=1
        if not(place):
            if p == I: groupes.append([i])
            else: groupes.append([p,i])
    return(groupes)
```

# Partie III. Algorithme randomisé pour la coupe minimum

**Question 14** Pour la fonction `coupeMinimumRandomisee`, on suit l'algorithme décrit par l'énoncé.
Ligne 4, on crée une partition de $n$ singletons de $[\![n]\!]$. La variable `cardP` désigne le nombre de groupes. La variable $m$ désigne le nombre de liens non encore marqués. Dans le déroulement du programme les $m$ liens non marqués sont placés au début de la liste des liens `liens`.
Ligne 7 et 8, on choisit au hasard un lien non marqué $[i, j]$. Lignes 9 et 10, on récupère les représentants du groupe contenant $i$ et du groupe contenant $j$. Lignes 11 à 13, si $i$ et $j$ n'appartiennent pas au même groupe, on fusionne les deux groupes. Ligne 14, on permute le lien étudié avec le dernier des liens non marqués. Ligne 15, on décrémente $m$, ainsi le lien $[i, j]$ devient un lien marqué. La boucle conditionnelle des lignes 6 à 15 s'arrête dès que le nombre de groupes est égal à 2 ou que tous les liens ont été marqués. Dans la boucle des lignes 18 à 22, on effectue des fusions jusqu'à obtenir exactement deux groupes si ce n'est pas le cas. Ligne 23, on retourne le tableau `parent` codant les deux groupes construits.

```
1  def coupeMinimumRandomisee(reseau):
2      n = reseau[0]; liens = reseau[1]
3      m = len(liens)
4      parent = creerPartitionEnSingletons(n)
5      cardP = n
6      while m > 0 and cardP > 2:
7          p = random.randint(0,m)
8          (i,j) = liens[p]
9          a = representant(parent,i)
10         b = representant(parent,j)
11         if a != b:
12             fusion(parent,i,j)
13             cardP -= 1
14         liens[p],liens[m-1] = liens[m-1],liens[p]
15         m -= 1
16     print(parent)
17     g1 = representant(parent,1); i = 1
18     while cardP > 2:
19         g2 = representant(parent,i)
20         if g2 != g1:
21             fusion(parent,g2,g1); cardP -= 1
22         i += 1
23     return(parent)
```

Au cours de l'exécution de la fonction `coupeMinimumRandomisee`, on fait au plus $(n-1)$ appels à la fonction `fusion` dont le coût est en $O(\alpha(n))$ (deux appels à la fonction `representant`). On passe au plus $m$ fois dans la boucle des lignes 6 à 15. Sans compter les appels à la fonction `fusion`, on a au plus 5 opérations élémentaires et deux appels à la fonction `representant` ce qui représente un coût en $O(m(\alpha(n)+5))$.
La complexité de la fonction `coupeMinimumRandomisee` est donc en $O((n+m)\alpha(n))$.

**Question 15** Pour compter le nombre de liens entre les différents groupes de la partition, on parcourt l'ensemble des liens $(i, j)$ (boucle des lignes 5 à 8). Si $i$ et $j$ ne sont pas dans le même groupe, c'est-à-dire si leurs représentants de groupe sont différents, on incrémente le compteur de liens (ligne 8). À la ligne 9, on retourne la valeur demandée.

```
1  def tailleCoupe(reseau,parent):
2      liens = reseau[1]
3      m = len(liens)
4      compteur = 0
5      for k in range(m):
6          (i,j) = liens[k]
7          if representant(parent,i) != representant(parent,j):
8              compteur += 1
9      return(compteur)
```

# B- Programmation SQL

**Question 16** Parmi tous les liens de la table `LIENS`, on retourne l'identifiant `id2` si `id1 = x`.

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

**Question 17** On fait la jointure entre les tables `LIENS` et `INDIVIDUS` avec le critère `id=id2`. On en extrait les noms et prénoms pourvu que `id1` = `x`, c'est-à-dire pourvu que `id2` soit un ami de `x`.

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

**Question 18** On fait la jointure entre la tables `LIENS` et elle-même avec le critère `a.id2` = `b.id1`. Les individus `a.id1` et `b.id2` ont donc pour amis commun l'individu `a.id2` = `b.id1`. On extrait de cette table tous les identifiants `b.id2` tels que a.id1 = x.

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