I. RÉSEAUX SOCIAUX

S1. Exemple des réseaux A et B :

```
reseauA=[5,[[0,1],[0,2],[0,3],[1,2],[2,3]]]
reseauB=[5,[[0,1],[1,2],[1,3],[2,3],[2,4],[3,4]]]
```

S2. La fonction `creerReseauVide(n)` :

```
creerReseauVide=lambda n:[n,[]]
```

S3. La fonction `estUnLienEntre(paire,i,j)` :

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

S4. La fonction `sontAmis(reseau,i,j)` :

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

La fonction, dans le pire des cas, fait m appels de la fonction `estUnLienEntre`. Cette dernière réalise 4 comparaisons. Ainsi, la complexité de `sontAmis` est $\mathcal{O}(m)$.

S5. La procédure `declareAmis(reseau,i,j)` :

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

La complexité est $\mathcal{O}(m)$. C'est la même complexité que la fonction `sontAmis`. Il y a éventuellement un affectation en plus, ce qui ne changepas la complexité asymptotique.

S6. La fonction `listeDesAmisDe(reseau,i)` :

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

Il y a $2m$ comparaisons et au pire m affectations. La complexité est $\mathcal{O}(m)$.

II. PARTITIONS

S7. Les représentants des groupes de la filiation A sont : 1,3,4 et 5.

```
parentA=[5,1,1,3,4,5,1,5,5,7]
```

Les représentants des groupes de la filiation B sont : 3,7 et 9.

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

S8. La fonction `creerPartitionEnSingletons(n)` :

```
creerPartitionEnSingletons=lambda n:list(range(n))
```

S9. La fonction `representant(parent,i)` :

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

Le cas le pire est quand tous les éléments forment une suite de filiation. Ainsi, en partant du dernier, il faudra remonter toute la filiation jusqu'au représentant ce qui fait $n-1$ étapes. Par exemple :
`parent=[1,2,3,...,n-2,n-1,n-1]`
La complexité est $\mathcal{O}(n)$.

S10. La fonction `fusion(parent,i,j)` :

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

S11. On peut opérer par :

appels	rep. de 0	nb d'opératons
`fusion(parent,0,1)`	1	1
`fusion(parent,0,2)`	2	2
`fusion(parent,0,3)`	3	3
⋮	⋮	⋮
`fusion(parent,0,n-1)`	n-1	n-1

Le nombre des opérations élémentaires est successivement 1,2,...,n-1, ce qui donne un nombre total d'opérations équivalent à $\frac{n^2}{2}$.

S12. La nouvelle fonction `represent(parent,i)` :

```
def representant(parent,i):
    p=i
    while parent[p]!=p: p=parent[p]
    while i!=p:
        q=parent[i]
        parent[i]=p
        i=q
    return p
```

Le nombre d'opérations est doublé, ce qui ne change pas la complexité : $\mathcal{O}(n)$.

S13. La fonction `listeDesGroupes(parent)` :

```
def listeDesGroupes(parent):
    L=[[0]]
    for i in range(1,len(parent)):
        cond=True
        for e in L:
            if representant(parent,e[0])
                        ==representant(parent,i):
                e.append(i)
                cond=False
                break
        if cond: L.append([i])
    return L
```

L'idée est de parcourir tous les éléments l'un après l'autre. Puis de regarder pour chaque élément considéré s'il a le même parent que l'un des premiers éléments de chaque groupe déjà créé. Si oui, il suffit de l'ajouter au groupe correspondant ; si non, il convient de créer un nouveau groupe singleton partir de cet élément.

III. Algorithme randomisé pour la coupe

S14. La fonction `coupeMinimumRandomisee(reseau)` :

```
1  def coupeMinimumRandomisee(reseau):
2      P=creerPartitionEnSingletons(reseau[0])
3      lien=1*reseau[1]
4      while len(listeDesGroupes(P))>2
5                           and len(lien)>0:
6          i=random.randint(0,len(lien)-1)
7          l=lien.pop(i)
8          if representant(P,l[0])
9                           !=representant(P,l[1]):
10             fusion(P,l[0],l[1])
11     L=listeDesGroupes(P)
12     k=len(L)
13     for i in range((k-2)//2):
14         fusion(P,L[0][0],L[i+1][0])
15         fusion(P,L[-1][0],L[-2-i][0])
16     if k%2==1: fusion(P,L[0][0],L[k//2][0])
17     return listeDesGroupes(P)
```

Explications du script :

- ligne 3 : on fait une copie de lien du réseau
- ligne 4 : l'instruction `len(listeDesGroupes(P))` permet de connaître le nombre de groupes (on pourrait une fonction dédiée)
- ligne 6 : on choisit de retirer le lien traité plutôt que de le marquer, ce qui justifie le test `len(lien)>0` pour déterminer s'il reste des lien à considérer.
- lignes 11 et 12 : on reforme les groupes existant.
- lignes 13 à 15 : on fusionne les groupes deux à deux en partant des deux extrémités (l'objectif étant d'harmoniser les des deux groupes restant)
- ligne 16 : dans le cas où il restait un nombre impair de groupe (à la ligne 9), in reste une fusion à opérer.

Les complexités :

- La complexité de `creerPartitionEnSingletons` est en $\mathcal{O}(n)$.
- La complexité de `listeDesGroupes` est en $\mathcal{O}(n)$.
- La boucle while (ligne 4) est de longueur m.
- La complexité de `fusion` est en $\mathcal{O}(1)$.
- La boucle for (ligne 13 à 16) donne lieu à au maximum $m-2$ appel de la fonction `fusion`.

Ainsi, la complexité de `coupeMinimumRandomisee` est en $\mathcal{O}(n+m\alpha(n))$.

On remarque que $\alpha(n)$ est variable en 1 et n opérations. Au maximum n une fois puis 1 ensuite. On devrait donc pouvoir conclure que la complexité globale est en $\mathcal{O}(n+m)$.

S15. La fonction `tailleCoupe` :

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