I. Réseaux sociaux

1 On peut proposer par exemple la représentation suivante pour le réseau A

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

et la suivante pour le réseau B

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

> Dans les deux représentations ci-dessus, les listes représentant les liens d'amitiés sont triées par ordre lexicographique croissant. C'est essentiellement pour vérifier que l'on n'oublie personne, mais l'énoncé précise clairement que ce n'est pas une obligation. Par conséquent, n'importe quel ordre serait accepté dans la liste et dans chaque couple (les listes `[0,1]` et `[1,0]` représentent le même lien).

2 S'il n'y a aucune relation d'amitié dans le réseau, c'est que la seconde liste de la représentation est vide. Il suffit donc d'écrire la fonction suivante.

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

3 Cette fonction peut s'écrire de la façon suivante :

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

> Le test peut s'écrire un peu plus rapidement à l'aide de l'opérateur `==` qui fonctionne aussi pour tester l'égalité de deux listes. On peut donc écrire plus directement
>
> ```
> def estUnLienEntre(paire,i,j):
>    return (paire == [i,j]) or (paire == [j,i])
> ```
>
> Les paragraphes de la forme
>
> ```
> if un_test:
>     return True
> else:
>     return False
> ```
>
> sont maladroits et à proscrire. Il n'y a rien d'incorrect en terme de syntaxe, mais cela revient au même d'écrire en une ligne
>
> ```
> return un_test
> ```

4 Il suffit de passer en revue toutes les paires de la seconde liste du réseau et de vérifier s'il s'agit d'un lien entre `i` et `j`. Si c'est le cas au moins une fois, on termine immédiatement l'exécution en renvoyant `True`, sinon on renvoie `False` à la fin de la boucle. Cela peut s'écrire de la manière suivante :

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

Dans le pire des cas, il n'y a pas de lien entre `i` et `j` et il faut alors parcourir toute la liste `reseau[1]` avant de conclure. Chaque étape de la boucle nécessite une application de `estUnLienEntre` qui s'exécute en temps constant. Par conséquent,

La fonction `sontAmis` s'exécute en $O(m)$ opérations élémentaires.

> La syntaxe « `x in L` » s'évalue en un booléen qui vaut `True` si et seulement si l'élément `x` apparaît dans la liste `L`. On aurait pu envisager d'écrire
>
> ```
> def sontAmis(reseau,i,j):
>     return ([i,j] in reseau[1]) or ([j,i] in reseau[1])
> ```
>
> ce qui serait plus court et plus élégant en terme d'écriture. Toutefois, il y a de grandes chances que cette écriture soit sanctionnée car ne respectant pas les conditions de l'énoncé (qui donne une liste précise des intructions utilisables sur les listes et interdit les autres).

5 Pour écrire cette procédure, on commence par vérifier s'il existe un lien entre `i` et `j`. Si ce n'est pas le cas, il suffit d'ajouter `[i,j]` à la liste `reseau[1]`.

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

L'utilisation de `sontAmis` nécessite dans le pire cas $O(m)$ opérations élémentaires. Le test est éventuellement suivi de l'ajout d'un élément dans une liste qui est une opération de coût constant. Au final,

La procédure `declareAmis` s'exécute en $O(m)$ opérations élémentaires.

6 On pourrait utiliser la fonction `sontAmis` et appliquer cette fonction pour chaque valeur de `j` dans $[\![1 ; n]\!]$ mais cette solution est coûteuse car elle nécessite de parcourir plusieurs fois la même liste `reseau[1]`. La solution suivante est plus efficace car elle ne nécessite qu'un parcours :

- pour chaque élément de `reseau[1]`, on cherche si l'un des deux éléments de la liste est `i` ;
- lorsque c'est le cas, on ajoute l'autre élément à une liste initialement vide.

> Cette fonction nécessite qu'il n'y ait qu'une seule liste représentant chaque lien d'amitié dans `reseau[1]`, ce qui semble être une hypothèse implicite de l'énoncé. Sinon, on court le risque d'avoir des occurrences multiples d'un même ami `j` dans la liste renvoyée.

La fonction décrite ci-dessus s'écrit de la façon suivante :

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

Cette fonction parcourt une fois la liste `reseau[1]` et exécute pour chacun d'entre eux au plus quatre opérations élémentaires. Par suite,

La fonction `listeDesAmisDe` s'exécute en $O(m)$ opérations élémentaires.

> Si l'on utilise la solution décrite initialement (vers laquelle les questions précédentes ont tendance à orienter le candidat), on utilise $n-1$ fois la fonction `sontAmis` de coût $O(m)$. On obtient une solution de coût $O(m\,n)$, ce qui est donc moins efficace que la solution proposée et serait certainement sanctionné.

II. Partitions

7 Les deux partitions de l'énoncé sont représentées par les deux tableaux suivants :

`parentsA` =

0	1	2	3	4	5	6	7	8	9
5	1	1	3	4	5	1	5	5	7

`parentsB` =

0	1	2	3	4	5	6	7	8	9
3	9	0	3	9	4	4	7	1	9

Dans la première représentation, les groupes sont $\{0,5,7,8,9\}$, $\{4\}$, $\{1,2,6\}$ et $\{3\}$, et ont pour représentants respectifs 5, 4, 1 et 3. Dans la seconde, les groupes sont $\{1,4,5,6,8,9\}$, $\{7\}$ et $\{0,2,3\}$ avec pour représentants respectifs 9, 7 et 3.

8 Une partition en singletons est représentée par un tableau où chaque élément est son propre représentant. On peut donc proposer la fonction suivante :

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

> Le langage python permet également la syntaxe raccourcie suivante
>
> ```
> def creerPartitionEnSingletons(n):
>     return list(range(n))
> ```
>
> ou encore, en utilisant les listes par compréhension
>
> ```
> def creerPartitionEnSingletons(n):
>     return [i for i in range(n)]
> ```
>
> mais dans le doute, il vaut peut-être mieux se contenter d'utiliser l'une des méthodes suggérées par l'énoncé pour la création de listes. On aurait donc pu définir une liste vide puis utiliser des `append` successifs.

9 Le représentant d'un individu est égal à lui-même s'il est son propre parent, ou au représentant de son parent sinon. Cela justifie l'écriture récursive suivante :

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

Dans la fonction ainsi définie, la variable `i` ne peut prendre que des valeurs deux à deux distinctes. Il ne peut donc y avoir qu'au plus n appels récursifs et ainsi,

> La fonction `representant` s'exécute en $O(n)$ opérations élémentaires dans le pire cas.

Un exemple de tableau pour lequel cette borne est atteinte est donné par

$$\texttt{parents} = \begin{array}{|c|c|c|c|c|c|c|c|}\hline 0 & 1 & 2 & 3 & \cdots & n-3 & n-2 & n-1 \\ \hline 1 & 2 & 3 & 4 & \cdots & n-2 & n-1 & n-1 \\ \hline \end{array}$$

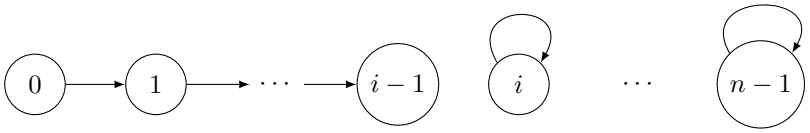
pour lequel la recherche du representant de 0 nécessite de l'ordre de n opérations élémentaires (il y a exactement n appels récursifs).

> Une autre solution pour trouver le représentant du groupe contenant `i` consiste, comme l'indique l'énoncé, à remonter de parents en parents jusqu'à tomber sur un élément qui est son propre parent. Ce travail peut s'effectuer à l'aide d'une boucle `while`, ce qui donne la fonction suivante :
>
> ```
> def representant(parent,i):
>     while i!=parent[i]:
>         i=parent[i]
>     return i
> ```
>
> La complexité reste identique à celle de la solution récursive si bien que les deux solutions sont envisageables. On notera cependant dans cette dernière version l'utilisation d'une particularité de Python, à savoir la possibilité de modifier la valeur de l'un de ses arguments qui n'est pas une liste (ici c'est `i`). Cette possibilité est loin d'être offerte par tous les langages de programmation, à cause de tous les effets de bords qu'elle est susceptible d'engendrer. Ici, la modification n'est que temporaire : une fois le calcul du représentant terminé, la variable `i` retrouve la valeur qu'elle avait au départ.

10 Le principe est donné par l'énoncé, il n'y a plus qu'à le traduire en Python.

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

11 Considérons la partition en n singletons de $[\![n]\!]$ et effectuons la suite d'instructions « `fusion(parent,0,i)` » pour `i` variant de 1 à $n-1$. Par récurrence immédiate sur i, à la fin de l'instruction `fusion(parent,0,i-1)`, la structure filiale est dans l'état



L'exécution de `fusion(parent,0,i)` nécessite alors successivement

- le calcul du représentant `p` de `0`, qui s'effectue en $O(i)$ opérations élémentaires (il faut remonter les parents jusqu'au nœud $i-1$ qui est son représentant) ;
- le calcul du représentant `q` de `i`, qui n'est autre que lui-même à cet instant. Cette étape est donc de coût constant ;
- enfin la modification de la valeur de `parent[p]` qui est également de coût constant.

Par conséquent, l'instruction `fusion(parent,0,i)` a un coût de l'ordre de $O(i)$ et le coût total des $n-1$ fusions est de l'ordre de

$$O\left(\sum_{i=1}^{n-1} i\right) = O\left(\frac{n(n-1)}{2}\right) = O(n^2)$$

On a donc bien ce que l'on voulait.

Pour une partition de $[[n]]$ en n singletons, la suite de $n-1$ instructions

`fusion(parent,0,1)` ; `fusion(parent,0,2)` ; $\cdots$; `fusion(parent,0,n-1)` ;

nécessite de l'ordre de n^2 opérations élémentaires.

12 Utilisons là encore la programmation récursive pour rédiger cette fonction. Si `i` est son propre parent, il n'y a rien d'autre à faire que renvoyer `i`. Sinon, on commence par calculer par un appel récursif le représentant `j` du père de `i`, puis on modifie le père de `i` en lui attribuant la valeur trouvée. Enfin, on renvoie la valeur `j`.

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

On notera que la modification des représentants des ancêtres de `i` se fait alors pendant les appels récursifs et donc avant celle du père de `i`.

> Si l'on n'est pas à l'aise avec la programmation récursive, une solution plus élémentaire consiste à rajouter dans la fonction en remarque de la question 9 une liste initialement réduite à `i`, à laquelle on va rajouter tous les indices rencontrés sur le chemin entre `i` et son représentant. Il n'y a plus alors qu'à faire une modification de `parent` pour tous les éléments de cette liste une fois que celle-ci est constituée. Cela donne la fonction suivante :
>
> ```
> def representant(parent,i):
>     ancetres=[i]
>     while i!=parent[i]:
>         i=parent[i]
>         ancetres.append(i)
>     for x in ancetres:
>         parent[x]=i
>     return i
> ```

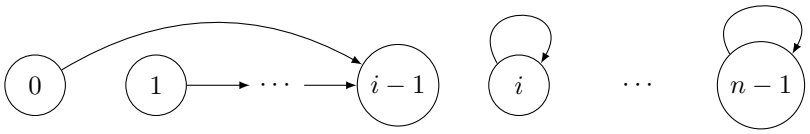
Cette optimisation peut être considérée comme « gratuite » pour essentiellement deux raisons :

- les appels récursifs effectués pour la recherche du représentant ne peuvent être évités, il s'agit d'un temps de calcul incompressible dans ce programme ;
- l'instruction rajoutée `parent[i]=j` est de coût constant. En particulier, **l'ordre de grandeur** de la complexité de la fonction `representant` est inchangé (même si en pratique, le coût est doublé puisqu'on rajoute une opération élémentaire à chaque appel récursif).

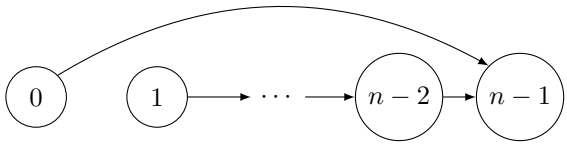
En algorithmique, les complexités sont quasi-systématiquement données par leur ordre de grandeur, et on ne prête que peu d'attention aux constantes masquées par les $\mathrm{O}(\cdot)$. De ce point de vue, on peut donc dire que

L'optimisation de la structure filiale peut être considérée comme « gratuite ».

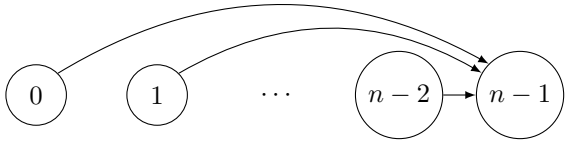
Examinons l'exemple de la question 11 sous l'effet de l'optimisation de la question 12. On peut facilement vérifier par récurrence sur i que la structure filiale après l'instruction `fusion(parent,0,i)` est de la forme



Le calcul du représentant de 0 lors de l'instruction `fusion(parent,0,i)` ne prend déjà plus qu'un coût constant, indépendant de i. Les $n-1$ fusions se font pour un coût $\mathrm{O}(n)$ à l'issue desquelles la structure filiale est de la forme



Notons qu'à cet instant, malgré l'optimisation, il reste des éléments (en l'occurrence 1) pour lesquels la recherche du représentant prend un coût de l'ordre de $\mathrm{O}(n)$ bien qu'il n'y ait qu'un seul groupe, et donc qu'un seul représentant. Toutefois, si par exemple on effectue la recherche du représentant de 1, la technique de compression recompose la structure filiale sous la forme



et cette fois, tous les calculs des représentants se feront par la suite en temps constant !

13 Une méthode simple consiste à initialiser une liste `L` de n listes vides, puis à parcourir le tableau `parent`. On ajoute alors l'élément `i` à la liste `L[representant[i]]`. À la fin du parcours, les éléments ayant le même représentant sont dans la même liste, mais certaines listes sont vides. Il suffit alors d'effectuer un nouveau parcours de `L` pour en extraire les listes non vides. On obtient alors la fonction suivante :

```
def listeDesGroupes(parent):
    listes=[]
    n=len(parent)
    for i in range(n):
        listes.append([])
    for i in range(n):
        listes[representant(parent,i)].append(i)
    groupes=[]
    for x in listes:
        if x != []:
            groupes.append(x)
    return groupes
```

> Une erreur classique quand on initialise une liste de listes consiste à utiliser l'écriture
>
> `l = [[]]*n`
>
> On obtient bien une liste `l` de listes `l[0], l[1], ..., l[n-1]` en apparence, mais en pratique, toutes les listes contenues dans `l` sont la même liste. Une modification sur l'une d'entre elles est donc répliquée sur toutes les autres ! C'est ce qui justifie l'utilisation de la première boucle et l'initialisation de `listes` par une liste vide.
>
> On aurait pu utiliser une optimisation similaire à celle de la question 12 et, lors du calcul d'un représentant dans la deuxième boucle, rajouter non seulement i, mais tous les sommets visités lors de la recherche de son représentant à la liste `listes[representant(parent,i)]`. Toutefois, la version optimisée de la fonction `representant` permet d'avoir des coûts de recherche de temps constant pour tous les ancêtres de `i` après le calcul de son représentant, ce qui limite l'intérêt de l'optimisation ci-dessus.

III. Algorithme randomisé pour la coupe minimale

14 Pour rédiger l'algorithme, on commence par définir une fonction préliminaire qui permet de permuter deux éléments dans un tableau à partir de leurs indices de positions.

```
def permute(l,i,j):
    a=l[j]
    l[j]=l[i]
    l[i]=a
```

C'est cette fonction qui sera utilisée pour marquer les liens d'amitiés, en les positionnant vers la fin du tableau.

Pour réaliser les fusions de l'étape 4 de l'algorithme, on utilise la méthode suivante qui permet de passer d'un réseau de $k \geqslant 3$ groupes à un réseau de 2 groupes :

- on calcule les représentants de chaque individu en commençant par celui d'indice 1 ;
- dès qu'un individu n'est pas dans la classe de 0, on fusionne sa classe avec celle de 0 et on met à jour le compteur de classes.
- on s'arrête lorsque le compteur atteint la valeur 2.

> On notera que cette méthode est susceptible de faire en sorte que la classe de 0 soit plutôt peuplée et on pourrait préférer des fusions de classe un peu plus aléatoires. Mais puisque l'énoncé n'impose rien, autant faire simple.

On peut alors rédiger l'algorithme de l'énoncé comme suit. La variable `g` est utilisée pour compter le nombre de groupes. Elle décroît d'une unité à chaque fusion. La variable `l` sert à compter le nombre de liens non marqués, suivant l'indication de l'énoncé.

```
def coupeMinimumRandomisee(reseau):
    p=creerPartitionEnSingletons(reseau[0])
    g=reseau[0]            # Nombre initial de groupes
    l=len(reseau[1])       # Nombre initial de liens
    while g>=3 and l>0:
        k=random.randint(0,l-1)
        couple=reseau[1][k]
        a=representant(p,couple[0])
        b=representant(p,couple[1])
        if a!=b:
            fusion(p,a,b)
            g-=1
        permute(reseau[1],k,l-1)
        l-=1
    if g>=3:
        a=representant(p,0)
        b=1
        while g>=3:
            if representant(p,b)!=a:
                fusion(p,0,b)
                g-=1
            b+=1
    return p
```

Examinons maintenant la complexité de cette fonction.

- le premier appel à `creerPartitionEnSingletons` est de coût $O(n)$, les initialisations de `g` et `l` de coût constant ;
- la boucle `while` s'exécute au plus m fois, car `l` est initialisée à m et décroît d'une unité à chaque boucle. Au sein de cette boucle, on trouve au plus 4 appels à `representant` (un pour `a`, un pour `b` et deux autres dans la fusion éventuelle), le reste n'étant que des opérations de coût constant ;
- reste le coût de la dernière boucle `while`, qui s'exécute au plus n fois puisque b augmente d'une unité à chaque étape et ne peut excéder n car alors on n'aurait plus qu'un seul groupe. Elle effectue donc $O(n)$ appels à `représentant`.

Au final, le coût de l'algorithme est donc de

$$O(n) + mO(\alpha(n)) + nO(\alpha(n))$$

et ainsi **Le coût de `coupeMinimumRandomisee` est en $O((m+n)\alpha(n))$.**

> Si le graphe des liens d'amitiés est dense (c'est-à-dire si tout le monde est copain avec tout le monde), alors m est de l'ordre de n^2 et la complexité est de l'ordre de $O(n^2\alpha(n))$. Mais si le nombre de liens d'amitiés par personne est raisonnable (tout le monde n'est pas Justin Bieber et ses millions d'amis sur Facebook), alors m est de l'ordre de n et la complexité est plutôt de l'ordre de $O(n\alpha(n))$.

15 Pour effectuer ce comptage, on passe en revue tous les liens d'amitiés et on incrémente un compteur lorsqu'un couple lie deux personnes qui n'ont pas le même représentant, ce qui signifie qu'ils ne sont pas dans le même groupe.

```
def tailleCoupe(reseau,parent):
    nb_liens=0
    for c in reseau[1]:
        a=representant(parent,c[0])
        b=representant(parent,c[1])
        if a!=b:
            nb_liens+=1
    return nb_liens
```

16 Une requête convenable serait

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

> L'hypothèse de l'énoncé (précisément la symétrie de la table INDIVIDUS) est essentielle au fonctionnement de cette requête. En effet, si par exemple la table ne contient que les liens d'amitiés $(1,2)$ et $(2,3)$, la requête trouvera 3 comme seul ami de 2, et passera donc à coté de l'ami 1.

17 La requête est à peu près la même, mais contient cette fois une jointure.

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

18 Une idée simple consiste à faire une jointure sur deux copies de la table LIENS (avec un renommage à la clé pour éviter les confusions). La jointure contient des quadruplets de la forme (a, b, c, d) où a et b sont amis, de même que c et d. On cherche alors à récupérer les éléments d pour lesquels $a = x$ et $b = c$, ce qui s'obtient à l'aide d'une sélection basée sur ces deux conditions.

```
SELECT t2.id2
FROM LIENS AS t1 JOIN LIENS AS t2 ON t1.id2=t2.id1
WHERE t1.id1=x
```

Avec cette requête, le résultat peut contenir des doublons. Pour remédier à cela, on peut rajouter la précision `DISTINCT` juste après le `SELECT`. Cet outil n'est cependant pas explicitement au programme, donc très certainement facultatif.

Il est clair que x est dans la liste des individus renvoyés (du moins si x a au moins un ami) mais l'énoncé ne précise pas s'il faut l'enlever du résultat. Au besoin, on peut modifier la dernière instruction par

```
WHERE t1.id1=x AND t2.id2!=x
```

Il n'y a pas unicité de la réponse à cette question. Notamment, on pourrait commencer par sélectionner d'abord les amis de x (en récupérant donc le travail des questions précédentes) avant de faire la jointure. Cela donne par exemple la requête suivante (où la précision `DISTINCT` a justement été ajoutée)

```
SELECT DISTINCT id1
FROM (SELECT id2 AS id3 FROM LIENS WHERE id1=x)
      JOIN LIENS ON id3=id2
```

Une autre solution qui n'utilise pas de jointure consiste à utiliser l'opérateur IN qui fonctionne comme un test (mais qui n'est pas explicitement au programme comme le serait une jointure).

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