Cette épreuve est constituée de deux problèmes indépendants.

Pour ce sujet, **le langage de programmation utilisé sera Python**. Vous pourrez utiliser les fonctions **Python** de manipulation de listes ou de matrices suivantes :

- Création d'une liste de taille n remplie avec la valeur x : `li = [x] * n`.
- Obtention de la taille d'une liste `li` : `len(li)`.
- Si `li` est une liste de n éléments, on peut accéder au k^e élément (pour $0 \leq k < $ `len(li)`) avec `li[k]`. On peut définir sa valeur avec `li[k] = x`.
- Un élément `x` peut être ajouté dans une liste `li` à l'aide de `li.append(x)`. On considèrera qu'il s'agit d'une opération élémentaire.
- Les matrices sont des listes de listes, chaque sous-liste étant considérée comme une ligne de la matrice. Si `mat` est une matrice, elle possède `len(mat)` lignes et `len(mat[0])` colonnes.
- Création d'une matrice de n lignes et p colonnes, dont toutes les cases contiennent x : `mat = [[x for j in range(p)] for i in range(n)]`.
- On accède à (resp. modifie) l'élément de `mat` dans la i^e ligne et j^e colonne avec `mat[i][j]` (resp. `mat[i][j] = x`).

À moins de les redéfinir explicitement, l'utilisation de toute autre fonction sur les listes (`sort`, `index`, `max`, etc.) est interdite. On rappelle enfin qu'une fonction qui s'arrête sans avoir rencontré l'instruction `return` renvoie `None`.

Problème 1 : Points proches dans le plan

Ce problème, pouvant par exemple survenir dans le domaine de la navigation maritime, vise à déterminer, dans un nuage de points du plan, la paire de points les plus proches. Il est constitué de trois parties dépendantes.

Formellement, on suppose qu'on dispose de n points dans le plan $(M_0, M_1, \ldots, M_{n-1})$ dans un ordre quelconque pour le moment. Ils seront représentés en Python par deux listes de flottants de taille n : `coords_x` et `coords_y`, donnant respectivement les abscisses et les ordonnées des points. On dira ainsi que M_i est le point d'indice i, qu'il a pour abscisse $x_i =$ `coords_x[i]` et pour ordonnée $y_i =$`coords_y[i]`. On supposera que `coords_x` et `coords_y` sont des variables globales, qu'on ne modifiera jamais au cours de l'exécution de l'algorithme.

1 Approche exhaustive

On utilise la distance euclidienne définie par $d(M_i, M_j) = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}$.

► **Question 1** Écrire une fonction `distance(i, j)` qui renvoie la distance entre les points M_i et M_j. On utilisera la fonction `sqrt` après l'avoir importée.

► **Question 2** Rappeler sommairement comment sont stockés les flottants en mémoire. Quelle conséquence cela peut-il avoir sur le calcul de la distance ? On ignorera par la suite les problèmes d'approximation.

► **Question 3** Écrire une fonction `plus_proche()` qui renvoie, à l'aide d'une recherche exhaustive, le couple d'entiers des indices `i` et `j` des deux points les plus proches du nuage de points.

► **Question 4** Donner, en la justifiant sommairement, la complexité de la fonction précédente en fonction de n.

2 Quelques outils pour s'améliorer

On souhaite maintenant obtenir la distance entre les deux points les plus proches avec une meilleure complexité. Pour cela nous allons décrire un algorithme utilisant une méthode de type *diviser pour régner*. Cette partie introduit des fonctions utiles pour la mise en œuvre de cet algorithme.

On se donne la fonction suivante :

```
def tri(liste):
    n = len(liste)
    for i in range(n):
        pos = i
        while pos > 0 and liste[pos] < liste[pos-1]:
            liste[pos], liste[pos-1] = liste[pos-1], liste[pos]
            pos -= 1
```

► **Question 5** Que renvoie cette fonction ? Que fait-elle ? Le démontrer soigneusement en exhibant un invariant de boucle.

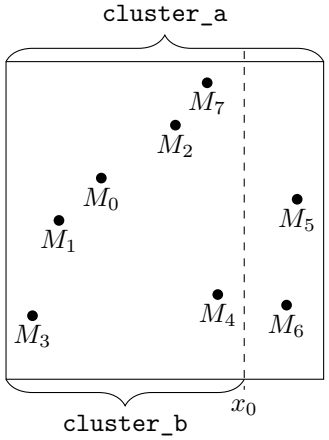
► **Question 6** Donner, en la démontrant, la complexité de la fonction `tri` en fonction de la taille de la liste donnée en paramètre.

► **Question 7** On souhaite trier une liste contenant des indices de points suivant l'ordre des abscisses croissantes. Que faudrait-il changer à la fonction `tri` ci-dessus pour qu'elle réalise cette opération ?

► **Question 8** Indiquer le nom d'un autre algorithme de tri plus efficace dans le pire des cas, ainsi que sa complexité. On ne demande pas de le programmer.

On admettra que l'on dispose de deux listes de n entiers `liste_x` (resp. `liste_y`) contenant les indices des points du nuage triés par abscisses croissantes (resp. par ordonnées croissantes). On supposera désormais que deux points quelconques ont des abscisses et des ordonnées distinctes.

Dans toute la suite, un sous-ensemble de points sera décrit par un *cluster*. Un cluster est une matrice de deux lignes contenant chacune les mêmes numéros correspondant aux numéros des points dans le sous-ensemble considéré. Dans la première ligne, les points sont triés par abscisses croissantes ; dans la seconde, ils sont triés par ordonnées croissantes. La figure 1 donne la représentation de deux clusters.




```
liste_x = [3, 1, 0, 2, 7, 4, 6, 5]
liste_y = [3, 6, 4, 1, 5, 0, 2, 7]

cluster_a = [liste_x, liste_y]

cluster_b = [ [3, 1, 0, 2, 7, 4],
              [3, 4, 1, 0, 2, 7] ]
```

FIGURE 1 – Représentation en Python de deux clusters

Pour être efficace, notre algorithme ne doit pas re-trier les listes des indices de points à chaque étape. Nous allons donc définir une fonction qui permet d'extraire des indices d'un cluster et former ainsi un nouveau cluster plus petit.

► **Question 9** Écrire une fonction `sous_cluster(cl, x_min, x_max)` qui prend en arguments un cluster `cl` et deux flottants `x_min` et `x_max`, et renvoie le sous-cluster des points dont l'abscisse est comprise entre `x_min` et `x_max` (au sens large). Cette fonction doit avoir une complexité linéaire *en la taille du cluster*.

► **Question 10** Écrire une fonction `mediane(cl)` qui prend en entrée un cluster `cl` contenant au moins 2 points et renvoie une abscisse médiane, c'est-à-dire que la moitié (au moins) des points a une abscisse inférieure ou égale à cette valeur, et la moitié (au moins) des points a une abscisse supérieure ou égale à cette valeur. Cette fonction doit avoir une complexité en $O(1)$.

3 Méthode sophistiquée

Le fonctionnement de l'algorithme utilisant une méthode de type *diviser pour régner* est illustré par la figure 2 :

1. Si le cluster contient deux ou trois points, on calcule la distance minimale en calculant toutes les distances possibles.
2. Sinon, on sépare le cluster en deux parties G et D qu'on supposera de tailles égales (éventuellement à un point près) suivant la médiane des abscisses, qu'on notera x_0.
3. Les deux points les plus proches sont soit tous les deux dans G, soit tous les deux dans D, soit un dans G et un dans D.
4. On calcule récursivement le couple le plus proche dans G et le couple le plus proche dans D. On note d_0 la plus petite des deux distances obtenues.
5. On cherche s'il existe une paire de points (M_1, M_2) telle que M_1 est dans G, M_2 dans D, et $d(M_1, M_2) < d_0$.
6. Si on en trouve une (ou plusieurs), on renvoie la plus petite de ces distances. Sinon, on renvoie d_0.

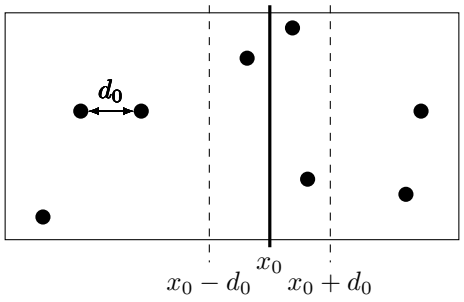


FIGURE 2 – Illustration du diviser pour régner

► **Question 11** Écrire une fonction `gauche(cl)` qui prend en argument un cluster `cl` contenant au moins deux points et renvoie le cluster constitué uniquement de la moitié (éventuellement arrondie à l'entier supérieur) des points les plus à gauche du cluster `cl`.

On suppose qu'on dispose d'une fonction `droite(cl)` qui renvoie le cluster contenant tous les autres points du cluster `cl` n'appartenant pas au cluster renvoyé par la fonction `gauche(cl)`.

► **Question 12** Justifier que l'on peut se contenter de chercher les points M_1 et M_2 de l'étape 5 de l'algorithme dans l'ensemble des points dont l'abscisse appartient à $I_0 = [x_0 - d_0, x_0 + d_0]$.

► **Question 13** Écrire une fonction `bande_centrale(cl, d0)` qui prend en argument un cluster `cl` et un réel `d0`, et renvoie le cluster des points dont l'abscisse est dans I_0. Cette fonction doit avoir une complexité linéaire en la taille du cluster.

► **Question 14** Montrer que deux points M_1 et M_2 (de l'étape 5 de l'algorithme) situés à une distance inférieure à d_0 se trouvent, dans la deuxième ligne du cluster (c'est-à-dire la ligne triée par ordonnées croissantes), séparés d'au plus 6 éléments.
On pourra montrer par l'absurde qu'un rectangle, à préciser, de dimensions $2d_0 \times d_0$ contient au plus 8 points.

► **Question 15** En déduire une fonction `fusion(cl, d0)` qui prend en entrée un cluster de points dont toutes les abscisses sont dans un intervalle $[x_0 - d_0, x_0 + d_0]$, et renvoie la distance minimale entre deux points du cluster si elle est inférieure à d_0, ou d_0 sinon. Cette fonction doit avoir une complexité linéaire en la taille du cluster `cl`. Vous justifierez cette complexité.

► **Question 16** Écrire une fonction récursive `distance_minimale(cl)` qui prend en argument un cluster et utilise l'algorithme décrit plus haut pour renvoyer la distance minimale entre deux points du cluster.

► **Question 17** Si on note n la taille du cluster `cl`, et $C(n)$ le nombre d'opérations élémentaires réalisées par la fonction `distance_minimale(cl)`, justifier que l'on a :

$$C(n) = 2C(n/2) + O(n)$$

► **Question 18** En déduire, en la démontrant, la complexité $C(n)$. On pourra se limiter au cas où n est une puissance de 2.

Problème 2 : Composantes connexes et biconnexes

4 Site Internet et bases de données

On s'intéresse dans cette partie à un site Internet d'échange de supports de cours entre enseignant · e · s de NSI. Chaque personne désirant proposer ou récupérer du contenu doit commencer par se créer un compte sur ce site et peut ensuite accéder à du contenu ou en proposer.

► **Question 19** Expliquer sommairement la différence entre Internet et le web.

► **Question 20** Expliquer deux conséquences du règlement général sur la protection des données (RGPD) sur le site Internet.

Ce site repose sur une base de données contenant en particulier trois tables.

- La table `comptes` possède un enregistrement par utilisateur ou utilisatrice, et ses attributs sont :
  - `id`, un identifiant numérique, unique pour chaque compte ;
  - `nom`, le nom de la personne possédant le compte ;
  - et d'autres informations, concernant le mot de passe, l'adresse mail, des préférences sur le site, etc., que nous ne détaillons pas ici.
- La table `ressources` possède un enregistrement par document téléversé sur le site. Ses attributs sont :
  - `id`, un identifiant numérique, unique pour chaque ressource ;
  - `owner`, l'identifiant de la personne ayant créé la ressource ;
  - `titre`, une chaine de caractères décrivant la ressource ;
  - `type`, chaine de caractères pouvant être `cours`, `ds`, `tp` ou `td`.
- La table `chargement` mémorise chaque fois qu'un utilisateur télécharge une ressource sur le site. Ses attributs sont :
  - `date`, date du téléchargement, par exemple `'2021-02-28'` pour le 28 février 2021 (on peut utiliser des opérations de comparaison classiques avec ce format) ;
  - `id_u`, identifiant de l'utilisateur qui télécharge la ressource ;
  - `id_r`, identifiant de la ressource téléchargée.

Voici un extrait de chacune de ces tables :

comptes

id	nom	…
1	Ada Lovelace	…
4	Alan Turing	…
…	…	…

ressources

id	owner	titre	type
4	1	Machine à décalage	cours
13	4	Intelligence artificielle	td
…	…	…	…

chargement

date	id_u	id_r
'1931-06-29'	4	4
'2020-05-30'	27	458
…	…	…

▶ **Question 21** Écrire une requête SQL permettant de connaitre le nombre total de ressources de type `cours` présentes sur le site.

▶ **Question 22** Que fait la requête suivante : ?

```
SELECT ressource.titre, comptes.nom
FROM chargement
  JOIN ressources ON ressouces.id = chargement.id_r
  JOIN comptes ON comptes.id = chargement.id_u
ORDER BY chargement.date DESC
LIMIT 1
```

▶ **Question 23** Écrire une requête SQL qui permet de déterminer la liste des triplets (x, y, n), signifiant que la personne possédant l'identifiant x a téléchargé n fois des documents téléversés par la personne possédant l'identifiant y.

On définit le graphe non-orienté $G(V, E)$ où V est l'ensemble des identifiants de comptes sur le site et $E \subset V \times V$ l'ensemble des paires d'identifiants telles que le premier compte a déjà téléchargé des documents téléversés par l'autre et réciproquement. Ainsi, si $(x, y) \in E$, alors on doit avoir $(y, x) \in E$.

▶ **Question 24** Écrire une requête SQL qui renvoie la table des couples (x, y) de E.

5 Composantes connexes

L'objectif de cette partie est de déterminer les composantes connexes du graphe G défini à la partie précédente. Dans toute la suite, on notera $|X|$ le cardinal d'un ensemble X. On supposera que l'ensemble V est constitué de sommets numérotés par des entiers consécutifs commençant à 0, c'est-à-dire que $V = \{0, 1, \ldots, |V| - 1\}$. On dira que deux sommets x, y de V sont *voisins* lorsque $(x, y) \in E$.

La requête de la question 24 permet de récupérer le résultat sous forme d'une liste de tuple à deux valeurs. On souhaite avoir plutôt une représentation par listes d'adjacences, à savoir une liste de $|V|$ sous-listes, la i^e sous-liste contenant les voisins du sommet i. On illustre ces différentes représentations avec le graphe G_{ex} de la figure 3.

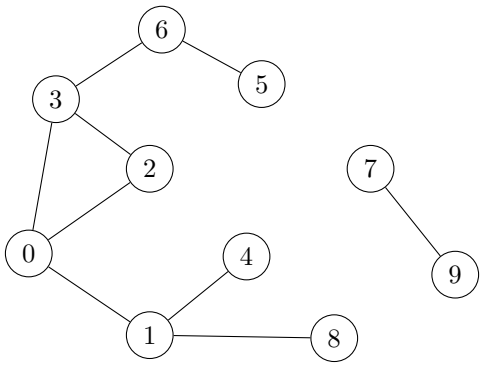


FIGURE 3 – Un graphe G_{ex}

Ce graphe serait obtenu à la question 24 sous la forme :

```
g_ex_a = [
    (0, 1), (0, 2), (0, 3), (1, 0), (1, 4), (1, 8),
    (2, 0), (2, 3), (3, 0), (3, 2), (3, 6),
    (4, 1), (5, 6), (6, 3), (6, 5),
    (7, 9), (8, 1), (9, 7)
    ]
```

Sa représentation par listes d'adjacences serait :

```
g_ex_b = [ [1, 2, 3], [0, 4, 8], [0, 3],
    [0, 2, 6], [1], [6], [3, 5] [9], [1], [7]
    ]
```

► **Question 25** Écrire une fonction `adjacences(n, li)` qui prend en argument un entier `n` correspondant à $|V|$ et `li`, une liste de couples correspondant à un ensemble E (comme par exemple `g_ex_a`) dans un ordre quelconque, et renvoyant la représentation du graphe $G(V, E)$ sous forme de listes d'adjacences (comme par exemple `g_ex_b`).

On se donne le programme suivant :

```
class Arbre():
    def __init__(self, sommet):
        self.sommet = sommet
        self.children = []

    def add_child(self, child):
        self.children.append(child)


def parcours(listes_adjacences):
    n = len(listes_adjacences)
    deja_vu = [False] * n

    def explorer(i):
        arbre = Arbre(i)
        voisins = listes_adjacences[i]
        for s in voisins:
            if not deja_vu[s]:
                deja_vu[s] = True
                arbre.add_child(explorer(s))
        return arbre

    res = []
    for i in range(n):
        if not deja_vu[i]:
            deja_vu[i] = True
            res.append(explorer(i))
    return res
```

► **Question 26** Quel est le type de la valeur renvoyée par la fonction `parcours` ? Appliquer à la main cette fonction sur la liste d'adjacence `g_ex_b` du graphe G_{ex} de la figure 3, et représenter la valeur de retour de cette fonction. Quel est le nom de ce parcours ?

► **Question 27** Montrer que la complexité de la fonction `parcours` est en $O(|V| + |E|)$. Dans toute la suite, on dira qu'un algorithme ayant cette complexité est *linéaire*.

► **Question 28** Rappeler la définition de la connexité d'un graphe.

► **Question 29** Écrire une fonction `connexe(listes_adjacences)` qui renvoie `True` si le graphe décrit par les listes d'adjacences `listes_adjacences` est connexe et `False` sinon.

► **Question 30** Écrire une fonction `composantes_connexes(p_graphe)` prenant en argument `p_graphe` le graphe obtenu avec la fonction `parcours` et renvoie les composantes connexes sous forme de liste de listes de sommets.

► **Question 31** Quelle est la limitation liée au fait que la fonction `explorer`, codée en Python, est récursive ?

Dans toute la suite, lorsqu'une fonction est demandée, on pourra utiliser ou non une fonction récursive, au choix des candidat.e.s.

6 Graphes biconnexes

On suppose dans cette partie que G est un graphe connexe. Si $\forall i \in [0;k]$ $x_i \in V$, on appelle *chaine* une suite finie $(x_0, x_1, \ldots, x_k)$ telle que pour tout i, on ait $(x_i, x_{i+1}) \in E$. Cette chaine est un *cycle* lorsque $x_0 = x_k$, et c'est de plus un *cycle élémentaire* si tous les sommets $x_0, \ldots, x_{k-1}$ sont distincts deux à deux. On dit que $G(V, E)$ est *biconnexe* lorsque :

- $|V| = 1$;
- $|V| = 2$, $V = \{a, b\}$ et $(a, b) \in E$;
- ou $|V| \geq 3$ et pour toute paire $(x, y) \in V^2$, il existe un cycle élémentaire contenant x et y.

► **Question 32** Montrer qu'un graphe biconnexe est également connexe.

► **Question 33** Donner un exemple de graphe connexe mais pas biconnexe.

On dit qu'un sommet x de G est un *point d'articulation* lorsque le graphe G privé du sommet x (et des arêtes issues de x) n'est plus connexe. Notre objectif dans cette partie est de montrer la propriété suivante si $G(V, E)$ possède au moins 3 sommets :
$G(V, E)$ est biconnexe si et seulement si G ne possède pas de point d'articulation.

► **Question 34** Sur le graphe G'_{ex} de la figure 4, donner les points d'articulations.

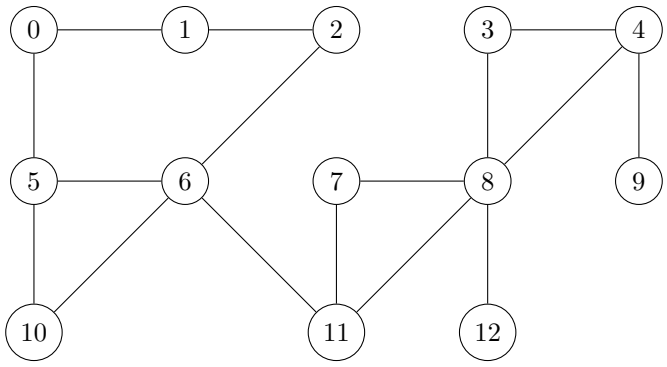


FIGURE 4 – Le graphe connexe G'_{ex}

Dans toute la suite, on considère un graphe G ayant au moins 3 sommets.

► **Question 35** Soit $G(V, E)$ possédant un point d'articulation. Montrer que G n'est pas biconnexe.

► **Question 36** Inversement, supposons $G(V, E)$ un graphe sans point d'articulation, et tel que $|V| \geqslant 3$. Considérons deux sommets x et y.

1. Justifier qu'il existe une chaine $(x_0 = x, x_1, \ldots, x_k = y)$ dans le graphe.
2. Montrer qu'il existe un cycle élémentaire contenant x_0 et x_1.
3. Pour $i \geqslant 1$, on suppose qu'il existe un cycle élémentaire C contenant x_0 et x_i. Montrer qu'il existe alors un cycle élémentaire contenant x_0 et x_{i+1}. On pourra distinguer deux cas selon que C contient ou non x_{i+1}.
4. En déduire que G est biconnexe.

► **Question 37** Expliquer comment on peut déterminer si un sommet particulier est un point d'articulation à l'aide d'un parcours en profondeur.

► **Question 38** En déduire un algorithme qui prend en entrée un graphe connexe décrit par ses listes d'adjacences, et détermine si ce graphe est biconnexe en utilisant la propriété précédente. On ne demande pas de programmer cet algorithme en Python. Quelle serait sa complexité en fonction des caractéristiques $|E|$ et $|V|$ du graphe ?

7 Algorithme efficace pour déterminer les points d'articulation

Dans cette partie, on détaille comment déterminer tous les points d'articulation d'un graphe $G(V, E)$ avec une complexité linéaire.

On modifie le programme `parcours` pour lui faire remplir et renvoyer une liste `prefixe` correspondant aux ordres d'appel de la fonction `explorer`. De plus, on suppose désormais que `listes_adjacences` décrit un graphe connexe, et on ne renverra qu'un seul arbre, issu de l'exploration à partir du sommet 0, et la liste `prefixe`. On utilise `nonlocal` pour que la variable `count` soit définie pour toute la fonction `parcours` et pas uniquement au sein de la fonction `explorer`.

```
def parcours(listes_adjacences):
    n = len(listes_adjacences)
    deja_vu = [False] * n
    prefixe = [-1] * n
    count = 1

    def explorer(i):
        nonlocal count
        prefixe[i] = count
        count += 1
        arbre = Arbre(i)
        voisins = listes_adjacences[i]
        for s in voisins:
            if not deja_vu[s]:
                deja_vu[s] = True
                arbre.add_child(explorer(s))
        return arbre

    deja_vu[0] = True
    return explorer(0), prefixe
```

► **Question 39** Donner les valeurs de la liste `prefixe` renvoyée par le programme ci-dessus si on l'applique sur le graphe G'_{ex} de la figure 4. On supposera que les voisins sont rangés par ordre croissant de leur numéro dans les listes d'adjacences.

► **Question 40** Soit G un graphe connexe dans lequel on réalise le parcours avec la fonction ci-dessus, et soit (i, j) une arête de G telle que `prefixe[i] < prefixe[j]`. Montrer que j est un descendant de i dans l'arbre.

Pour chaque sommet i, on note $\mathcal{V}(i)$ le voisinage de i, c'est-à-dire l'ensemble constitué de i et de ses voisins. Par extension, pour tout $A \subset V$, on notera $\mathcal{V}(A) = \cup_{i \in A} \mathcal{V}(i)$. En supposant réalisé un parcours dans l'arbre, on notera de plus $\mathcal{D}(i)$ l'ensemble des descendants de i dans l'arbre. Enfin, on définit $\texttt{ord}[i]$ par :

$$\texttt{ord}[i] = \min_{j \in \mathcal{V}(\mathcal{D}(i))} \texttt{prefixe}[j]$$

► **Question 41** Sur le graphe G'_{ex} de la figure 4, donner pour chaque sommet les valeurs de `ord[i]`, en se basant sur les valeurs obtenues à la question 39.

On admettra qu'un sommet i du graphe qui n'est pas la racine est un point d'articulation si et seulement si un de ses fils j vérifie $\texttt{ord}[j] = \texttt{prefixe}[i]$.
On supposera de plus qu'on dispose d'une fonction `calcule_ord(listes_adjacences)` qui renvoie la liste des `ord[i]` du graphe décrit par `listes_adjacences`, avec une complexité linéaire.

► **Question 42** Écrire une fonction `points_articulation(listes_adjacences)` qui renvoie la liste des points d'articulation d'un graphe. On fera attention à traiter la racine de l'arbre comme un cas particulier.

On définit une composante biconnexe d'un graphe G comme un sous-ensemble de sommets maximal (au sens de l'inclusion) qui est biconnexe.

▶ **Question 43** Sur le graphe G'_{ex} de la figure 4, donner la liste des composantes biconnexes.

▶ **Question 44** Décrire un algorithme qui renvoie les composantes biconnexes d'un graphe avec une complexité linéaire. On ne demande pas de programmer cet algorithme.