Éléments de correction de l'épreuve d'admissibilité 1

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.

```
from math import sqrt

def distance(i, j):
    xi, yi = coords_x[i], coords_y[i]
    xj, yj = coords_x[j], coords_y[j]
    return sqrt((xj-xi)**2 + (yj-yi)**2)
```

► **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.

Les flottants sont stockés en Python sous la forme $(-1)^s(1+x)2^k$ où s est un bit de signe qui vaut 0 ou 1, x est un nombre compris entre 0 et 1 stocké sur 52 bits, et k un nombre compris entre -1024 et 1023 (codé sur 11 bits).

Du fait d'une précision bornée, les calculs peuvent conduire à des erreurs d'approximation et dans le cas de figure, on pourrait imaginer des situations où les points considérés comme les plus proches (au sens de l'opération réalisée sur les flottants) ne sont pas les plus proches (au sens de l'opération mathématique).

▶ **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.

```
def plus_proche():
    n = len(coords_x)
    i_min, j_min = 0, 1
    for i in range(n):
        for j in range(i+1, n):
            if distance(i, j) < distance(i_min, j_min):
                i_min, j_min = i, j
    return i_min, j_min
```

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

Chaque appel à la fonction `distance` réalise un nombre borné d'opérations. On réalise deux tels appels pour chaque couple (i, j) avec $0 \leq i < j < n$, ce qui donne donc une complexité $O(n^2)$ pour la fonction `plus_proche()`.

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.

Cette fonction ne comporte pas d'instruction `return`, donc elle renvoie `None`.
Cette fonction trie la liste d'entiers donnée en argument par ordre croissant. En effet, on va démontrer qu'à la fin du i-ème passage dans la boucle for, les $i+1$ premiers éléments sont triés par ordre croissant. C'est vrai initialement (le premier élément de la liste est un élément trié). En supposant que les i premiers éléments sont triés par ordre croissant à la fin de la $(i-1)$-ème itération, l'itération suivante va considérer l'élément en position i, et l'intervertir avec ses voisins de gauche jusqu'à en rencontrer un plus petit ou arriver tout à gauche. Alors, par hypothèse de récurrence, tous les éléments à la gauche de l'élément placé seront inférieurs et leur ordre n'a pas changé, et tous les éléments à sa droite sont supérieurs et leur ordre relatif est inchangé. Ainsi, les $i+1$ premiers éléments sont désormais triés.

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

Chaque passage dans la boucle `while` réalise un nombre borné d'opérations, et on effectue au plus $i+1$ passages dans cette boucle (car `pos` décroit à chaque passage). La complexité est ainsi en $O(n^2)$.

► **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 ?

Il faudrait remplacer
`liste[pos] < liste[pos-1]` par
`coords_x[liste[pos]] < coords_x[liste[pos-1]]`.

► **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.

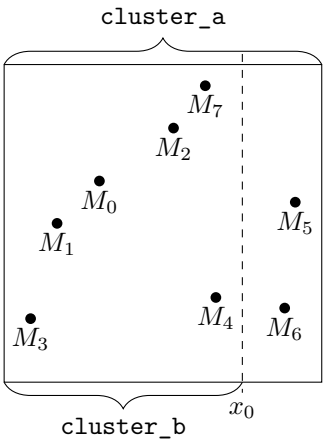
On pourrait utiliser un tri par fusion, dont la complexité est semi-linéaire ($O(n \ln n)$) dans le pire des cas.

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.
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*.

```
def sous_cluster(cl, x_min, x_max):
    liste_x = []
    for p in cl[0]:
        if x_min <= coords_x[p] <= x_max:
            liste_x.append(p)
```




```
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

```
    liste_y = []
    for p in cl[1]:
        if x_min <= coords_x[p] <= x_max:
            liste_y.append(p)
    return [liste_x, liste_y]
```

La complexité est bien respectée de manière évidente.

► **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 $O(1)$.

```
def mediane(cl):
    n = len(cl[0])
    return coords_x[cl[0][n//2]]
```

Ici encore la complexité demandée ($O(1)$) est bien respectée.

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.

► **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`.

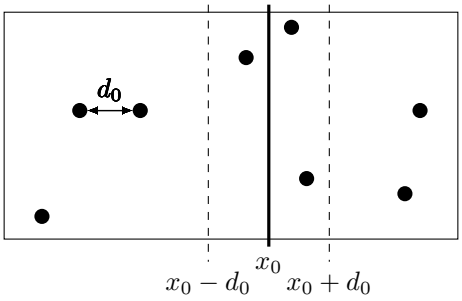


FIGURE 2 – Illustration du diviser pour régner

```
def gauche(cl):
    x_min = coords_x[cl[0][0]]
    return sous_cluster(cl, x_min, mediane(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 $I_0 = [x_0 - d_0, x_0 + d_0]$.

On note $M_1(x_1, y_1)$ et $M_2(x_2, y_2)$.
Supposons que le point M_1 soit situé à gauche de la droite d'équation $x = x_0 - d_0$. Alors la distance de M_1 à M_2 serait :

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \geq (x_2 - x_1) > d_0$$

Ainsi, le couple (M_1, M_2) ne peut pas être le couple de points les plus proches.
On procède de manière similaire si $x_2 > x_0 + d_0$.

► **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.

```
def bande_centrale(cl, d0):
    x = mediane(cl)
    return sous_cluster(cl, x-d0, x+d0)
```

► **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.

Soit $M_1(x_1, y_1)$ un point situé dans la bande I_0. Considérons le rectangle délimité par les droites d'équation $x = x_0 - d_0$, $x = x_0 + d_0$, $y = y_1$ et $y = y_1 + d_0$. Montrons par l'absurde que ce rectangle ne peut pas contenir plus de 7 points (en plus de M_1). Supposons qu'il en contienne 8, alors soit la moitié gauche de ce rectangle soit sa moitié droite contient 5 points. Cela signifie qu'on a 5 points du même côté de la droite d'équation $x = x_0$ dans un carré de côté d_0. En coupant ce carré en 4 quadrants, un des quadrants contient donc au moins deux points, et la distance maximale dans un quadrant est de $d_0/\sqrt{2} < d_0$. Cela contredit l'hypothèse que d_0 est la plus petite distance entre deux points du même côté.
Par ailleurs, un point plus proche de M_1 que la distance d_0 et d'ordonnée supérieure à M_1 serait nécessairement dans ce rectangle.

► **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.

```
def fusion(cl, d0):
    n = len(cl[0])
    d_min = d0
    for i in range(n):
        for j in range(1, 8):
            if i+j >= n:
                continue
            if distance(cl[1][i], cl[1][j]) < d_min:
                d_min = distance(cl[1][i], cl[1][j])
    return d_min
```

La boucle `for` interne a une complexité constante, tandis que la boucle `for` externe exécute n itérations, chacune de complexité constante, et est donc de complexité linéaire en n. Au total, la complexité de `fusion` est donc bien linéaire en la taille du cluster.

► **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.

```
def distance_minimale(cl):
    n = len(cl[0])
    if n == 2:
        return distance(cl[0][0], cl[0][1])
    if n == 3:
        d0 = distance(cl[0][0], cl[0][1])
        if distance(cl[0][0], cl[0][2]) < d0:
            d0 = distance(cl[0][0], cl[0][2])
        if distance(cl[0][1], cl[0][2]) < d0:
            d0 = distance(cl[0][1], cl[0][2])
        return d0
    d0 = distance_minimale(gauche(cl))
    dd = distance_minimale(droite(cl))
    if dd < d0:
        d0 = dd
    bande = bande_centrale(cl, d0)
    return fusion(bande, d0)
```

► **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 :

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

Si $n > 3$, le programme réalise deux tests puis utilise `gauche` et `droite`, puis `distance_minimale` deux fois sur deux clusters de taille divisée par deux, puis `bande_centrale` et `fusion` une fois chacun.
`gauche`, `droite`, `bande_centrale` et `fusion` ont une complexité linéaire en la taille du cluster, donc en $O(n)$.
En notant $C(n)$ la complexité de la fonction `distance_minimale`, les deux appels récursifs sont en $C(n/2)$ et on a donc bien : $C(n) = 2C(n/2) + O(n)$.

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

Dans la formule précédente, on écrira qu'on a $C(n) \leq 2C(n/2) + Kn$ où K est une constante ne dépendant pas de n.
On a bien $C(1)$ en $O(1)$.
En supposant que n s'écrit 2^i, on a $C(n) = C(2^i) = 2C(2^{i-1}) + K2^i$.
Par une récurrence immédiate, on a pour tout $k < i$: $C(n) = C(2^i) = 2^k C(2^{i-k}) + kK2^i$.
En particulier, avec $k = i - 1$, on trouve $C(n) = 2^{i-1}C(2) + (i-1)K2^i = C(2)n/2 + (\log_2(n) - 1)Kn = O(n \log_2(n))$.

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.

Internet désigne le réseau informatique international qui permet le transfert de données entre machines connectées à ce réseau. Le Web est une application de l'Internet qui permet, via un navigateur, d'accéder à un ensemble de ressources Web appelées URL.

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

Depuis l'entrée en vigueur du RGPD, des obligations pèsent sur les personnes hébergeant des informations nominatives. Parmi celles-ci, on peut citer :

- minimiser la quantité d'informations traitées,
- notifier aux personnes concernées toute violation des données les concernant,
- avoir une personne déléguée à la protection des données,
- permettre d'exercer un droit à l'oubli (effacement des données personnelles) et un droit à la portabilité (permettre de les obtenir dans un format structuré accessible).

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.

```
SELECT COUNT(*)
FROM ressources
WHERE type='cours'
```

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

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

Cette requête renvoie le nom de la personne ayant effectué le dernier téléchargement, ainsi que le titre de la ressource téléchargée.

▶ **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.

```
SELECT c.id_u AS x, r.owner AS y, COUNT(*) AS n
FROM chargement AS c
 JOIN ressources AS r ON c.id_r = r.id
GROUP BY r.owner, c.id_u
```

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.

Une possibilité :

```
SELECT DISTINCT c.id_u, r.owner
FROM chargement AS c
JOIN ressources AS r ON c.id_r = r.id
WHERE r.owner in ( SELECT DISTINCT c2.id_u
      FROM chargement AS c2
      JOIN ressources AS r2 ON c2.id_r = r2.id
      Where r2.owner = c.id_u)
AND c.id_u != r.owner
```

Une autre :

```
(SELECT sel1.id_u, sel1.owner FROM
(
    SELECT DISTINCT c.id_u, r.owner
    FROM chargement AS c
    JOIN ressources AS r ON c.id_r = r.id
) sel1
JOIN
(
    SELECT DISTINCT r.owner, c.id_u
    FROM chargement AS c
    JOIN ressources AS r ON c.id_r = r.id
) sel2

ON sel1.id_u = sel2.owner AND sel1.owner = sel2.id_u
```

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.

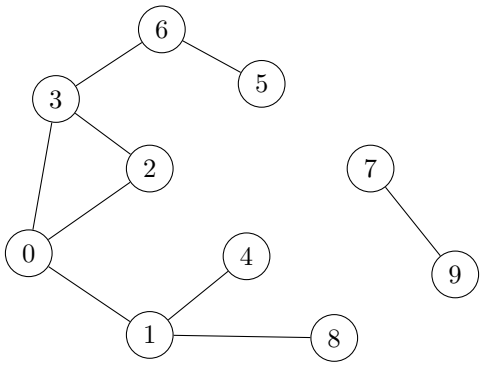


FIGURE 3 – Un graphe G_{ex}

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`).

```
def adjacences(n, li):
    res = [[] for i in range(n)]
    for a, b in li:
        res[a].append(b)
    return res
```

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)
```

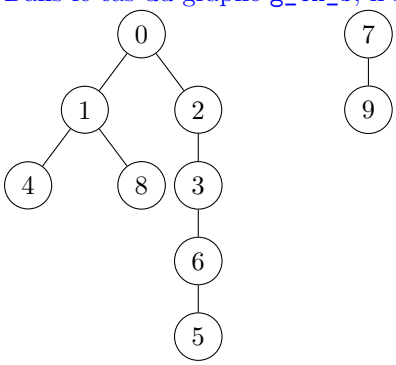
```
def parcours(listes_adjacences):
    n = len(listes_adjacences)
    deja_vu = [False] * n

    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 ?

La fonction renvoie une liste d'objets de type `Arbre`.
Dans le cas du graphe `g_ex_b`, il s'agit d'une liste de deux arbres :



Il s'agit d'un parcours en profondeur d'abord.

► **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*.

Cette fonction s'exécute avec une complexité en $O(|V| + |E|)$. En effet, la fonction `explorer` est appelée exactement une fois par sommet (avant de lancer `explorer` sur un sommet `i`, on place `deja_vu[i]` à `True` et cela empêche de relancer `explorer` sur ce sommet). Dans cet appel, on parcourt les voisins du sommet `i`, et pour chaque voisin on réalise un test et éventuellement un appel récursif. Le nombre total de tests est donc $|E|$, et le nombre total d'appels $|V|$. Enfin, sans les appels à la fonction `explorer`, la fonction `parcours` a une complexité en $O(|V|)$. La complexité totale est donc en $O(|V| + |E|)$.

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

Un graphe est connexe si pour toute paire (i, j) de sommets, il existe une chaine reliant i à j. C'est équivalent à n'avoir qu'un seul arbre renvoyé dans le parcours,

► **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.

```
def connexe(listes_adjacences):
    return len(parcours(listes_adjacences)) == 1
```

▶ **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.

```
def liste_sommets(arbre):
    res = [arbre.sommet]
    for fils in arbre.children:
        res += liste_sommets(fils)
    return res

def composantes_connexes(p_graphe):
    res = []
    for arbre in p_graphe:
        res.append(liste_sommets(arbre))
    return res
```

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

En Python, la pile d'appels de fonctions est forcément limitée (par défaut, à 1000 appels). Ainsi, on ne peut pas espérer effectuer un parcours dans un graphe si l'arbre devait avoir une profondeur supérieure à ce seuil.
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 $(x_0, x_1, \dots, x_k)$ 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 $x_0, \dots, x_{k-1}$ sont distincts deux à deux.
On dit que $G(V,E)$ est *biconnexe* lorsque :

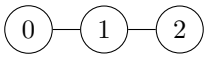
- $|V| = 1$;
- $|V| = 2$, $V = \{a, b\}$ et $(a, b) \in E$;
- ou $|V| \geq 3$ et pour toute paire $(x, y) \in V^2$, il existe un cycle élémentaire contenant x et y.

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

Si G est biconnexe, soient i et j deux sommets. Il existe un cycle élémentaire contenant i et j, donc en particulier il existe au moins une chaine reliant i et j, donc le graphe G est connexe.

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

Le graphe ci-dessous est bien connexe mais n'est 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.

Sur ce graphe, les sommets 4, 6, 8, et 11 sont des points d'articulation.
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.

Soient $G(V,E)$ et i un point d'articulation. Sans i, G n'est plus connexe, on peut donc trouver deux sommets x et y qui sont dans deux composantes connexes distinctes.
Ainsi, dans G, toute chaine allant de x à y contient nécessairement le sommet i. C'est contradictoire avec l'existence d'un cycle élémentaire contenant x et y : dans un tel cycle, le sommet i n'est présent qu'au plus une fois.

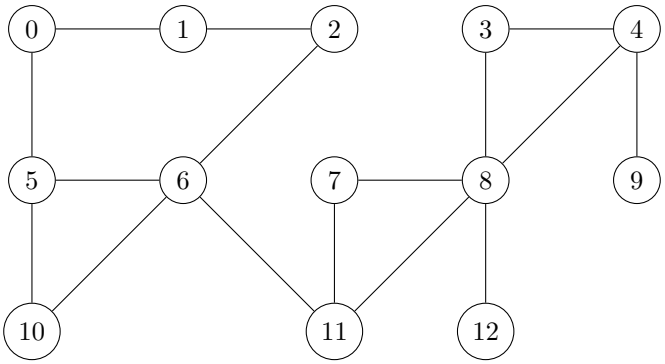


FIGURE 4 – Le graphe connexe G'_{ex}

► **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.

1. x et y font partie d'un graphe connexe, donc il existe une chaine reliant x à y.
2. Supposons que x_0 ne possède pour voisin que x_1. Alors x_1 serait un point d'articulation, car sa suppression empêcherait de relier x_0 aux autres sommets. Notons z un autre voisin de x_0. Comme x_0 n'est pas un point d'articulation, la suppression de x_0 conserve la connexité, donc il existe en particulier une chaine reliant z à x_1. Si on suppose complète cette chaine sans répétition et qu'on lui ajoute les arêtes (x_1, x_0) et (x_0, z), on obtient un cycle élémentaire.
3. Pour $i \geqslant 1$, on suppose qu'il existe un cycle élémentaire C contenant x_0 et x_i. Si ce cycle contient x_{i+1}, alors il y a un cycle élémentaire reliant x_0 et x_{i+1}.
   Si ce n'est pas le cas, notons C le cycle élémentaire contenant x_0 à x_i. Comme x_i n'est pas un point d'articulation, il existe une chaine allant de x_0 à x_{i+1} ne passant pas par x_i. Notons z le dernier sommet de cette chaine qui est également dans C. Alors on peut construire le cycle suivant : aller de x_0 à z en suivant C, puis de z à x_{i+1} en suivant la chaine décrite ci-dessus, puis de x_{i+1} à x_i car c'est une arête du graphe, puis de x_i à x_0 en utilisant la partie du cycle C qu'on n'a pas encore utilisée. Ce cycle est bien élémentaire.
4. En suivant le raisonnement par récurrence proposé, on trouve aussi qu'il existe un cycle élémentaire contenant x et y, et ce raisonnement est valable pour toute paire de points (x, y), donc le graphe est bien 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.

Soit i un point d'articulation. En lançant le parcours en profondeur depuis le sommet i, on obtient un arbre. Notons v_1 le premier voisin de i exploré dans le parcours. i possède au moins un autre sommet v_2 qui ne peut pas être exploré à partir de v_1, et la racine de l'arbre aura donc deux fils.
Inversement, si i n'est pas un point d'articulation, le sommet v_1 est relié à tout le reste du graphe, et son exploration ne pourra s'arrêter que si tout le reste du graphe a été exploré, ce qui signifie que i ne peut pas avoir d'autres fils.
i est donc un point d'articulation si et seulement si la racine du parcours à partir du sommet i possède plus d'un fils.

► **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 ?

On peut naïvement lancer un parcours en profondeur depuis chacun des sommets et appliquer le critère précédent. Cela donnerait une complexité en $O(|E| + |V|)$ pour chaque sommet (en réalité, comme le graphe est connexe on a $|V| - 1 \leq |E|$, donc on peut simplifier cette complexité en $O(|E|)$). Au total, la complexité serait donc en $O(|E||V|)$.

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.

`i`	0	1	2	3	4	5	6	7	8	9	10	11	12
`prefixe[i]`	1	2	3	10	11	5	4	8	9	12	6	7	13

► **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.

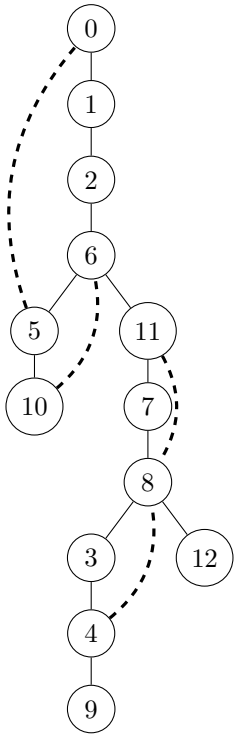
Si `prefixe[j] > prefixe[i]`, cela signifie qu'on explore `i` en premier. Toutes les explorations lancées depuis cette exécution donneront des nœuds descendants de i jusqu'à la fin de l'exécution de `explorer(i)`. Or comme il existe une arête (i, j), `explorer(i)` ne peut pas s'arrêter avant d'avoir placé le sommet `j`, qui est donc un descendant.

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.

Si on réalise le parcours, on obtient l'arbre ci-dessous, dans lequel on a fait figurer les arêtes manquantes en pointillés.



On peut donc compléter les valeurs du tableau ci-dessus :

i	prefixe[i]	ord[i]
0	1	1
1	2	1
2	3	1
3	10	9
4	11	9
5	5	1
6	4	1
7	8	7
8	9	7
9	12	11
10	6	4
11	7	4
12	13	9

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.

```
def points_articulation(listes_adjacences):
    n = len(listes_adjacences)
    arbre, prefixe = parcours(listes_adjacences)
    ord = calcule_ord(listes_adjacences)

    res = []

    def trouver_point(arbre):
        pere = arbre.sommet
        for ss_arbre in arbre.children:
            fils = ss_arbre.sommet
            if ord[fils] == prefixe[pere]:
                res.append(pere)
                break
        for ss_arbre in arbre.children:
            trouver_point(ss_arbre)

    for ss_arbre in arbre.children:
        trouver_point(ss_arbre)
    if len(arbre.children) > 1:
```

```
            res.append(arbre.sommet)
    return res
```

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.

Les composantes biconnexes sont : $\{\{0,1,2,5,6,10\},\{6,11\},\{11,7,8\},\{8,12\},\{8,3,4\},\{4,9\}\}$

► **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.

On reprend la recherche des points d'articulations. En même temps qu'on trouve les points d'articulation, on note les arêtes qui posent problème (quand i est un point d'articulation à cause de son fils j, on retient que l'arête (i,j) est particulière). On reprend alors un parcours complet de l'arbre, les sommets sont ajoutés à la composante bi-connexe de leur père, sauf quand on passe sur une arête particulière. Dans ce cas, le point d'articulation est ajouté à la composante connexe des deux extrémités, mais on crée alors deux composantes biconnexes distinctes.