Mini-projet : renforcement d'un réseau.

L'objectif de ce projet est d'implémenter des algorithmes permettant de mesurer la "fragilité" d'un réseau selon des critères définis plus bas, ainsi que des algorithmes permettant de manière automatique de renforcer les réseaux fragiles.

Les graphes considérés seront non orientés et construits à partir de données simplifiées du réseau RER/Métro de la RATP. Vous travaillerez obligatoirement à partir des fichiers de données fournis, mais vos tests devront aussi fonctionner sur d'autres réseaux.

Des jeux de tests sont également fournis sous la forme de doctests pour vous aider à vérifier vos fonctions ; votre travail sera évalué sur ces tests *et* sur d'autres jeux de tests aléatoires.

L'échéance est fixée au **dimanche 9 mai 2021 à 23h55**.

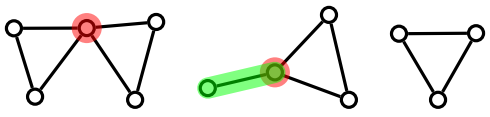
Avant de commencer, lisez intégralement le sujet et consultez les doctests donnés en exemple afin de ne pas partir dans une mauvaise direction ! !

Notions de base

On peut mesurer la fragilité d'un réseau grâce aux notions suivantes :

- un *point d'articulation* dans un graphe est un sommet dont la suppression augmente le nombre de composantes connexes ;
- un *pont* dans un graphe est une arête dont la suppression augmente le nombre de composantes connexes.

Par exemple, le graphe G_1 ci-dessous contient un point d'articulation et aucun pont. G_2 contient un pont et un point d'articulation, et G_3 ne possède ni pont ni point d'articulation.



G_1 G_2 G_3

Intuitivement, ces ponts et ces points d'articulation fragilisent le réseau, puisqu'un dysfonctionnement (correspondant à leur suppression) empêche deux parties du réseau d'être mutuellement accessibles.

Calcul de la fragilité d'un graphe

Détection des points d'articulation

Un algorithme efficace basé sur la construction de la forêt de parcours en profondeur permet de détecter les points d'articulations [1]. Le parcours en profondeur d'un graphe G à partir d'un ou plusieurs sommets arbitraires fournit une forêt F de parcours. Il oriente également les arêtes de G examinées lors du parcours et les classe en deux catégories :

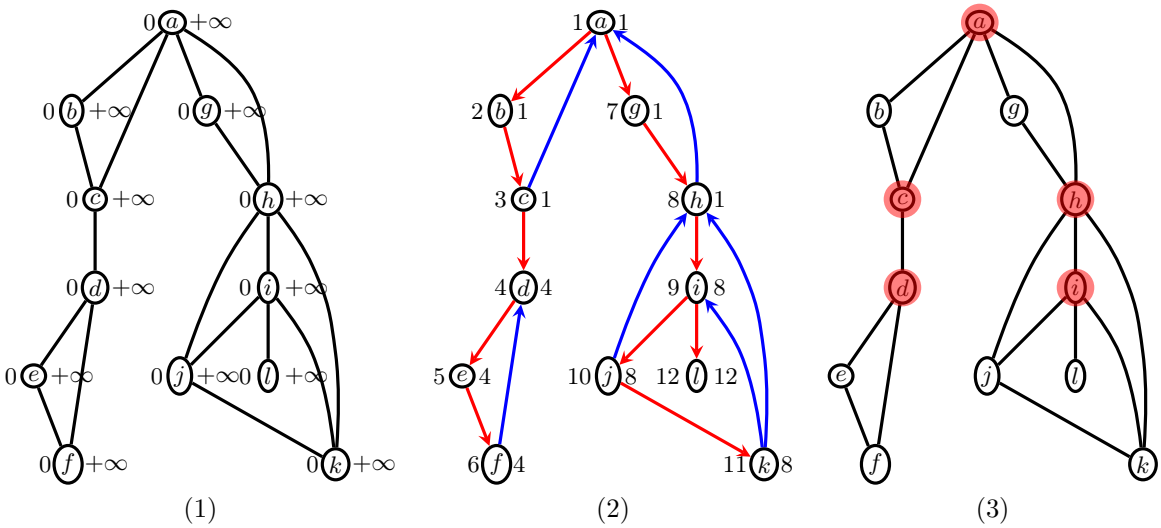
1. les *arcs d'arbre* sont les arêtes de F orientées conformément au parcours réalisé, et
2. les *arcs retour* n'appartiennent pas à F et nous renvoient vers un ancêtre déjà exploré.

Lors du parcours, en plus du parent de chaque sommet v de G, on stockera aussi :

1. début[v] : (initialement 0) la "date" de début de son exploration, qui indiquera aussi si v a déjà été visité. Cela correspond à numéroter les sommets dans l'ordre où on les visite.
2. ancêtre[v] : (initialement $+\infty$) la date de début d'exploration de l'ancêtre de v le plus lointain accessible en suivant un nombre arbitraire d'arcs d'arbre mais au plus un arc retour.

1. On admettra que les algorithmes proposés ici fonctionnent ; le lecteur curieux consultera Even [1] pour les démonstrations.

En guise d'exemple, voici (1) un graphe avec les structures de données auxiliaires avant le démarrage du parcours : pour chaque sommet v, on écrit à gauche début[v] et à droite ancêtre[v]. L'étape (2) montre le calcul au départ du sommet a d'un arbre de parcours en profondeur dont les arcs sont rouges ; les arcs retour sont en bleu, et on a mis à jour les deux numérotations.



L'étape (3) montre les cinq points d'articulation du graphe, identifiés comme suit :

1. si u est la racine d'un arbre de F, alors u est un point d'articulation si et seulement si son degré sortant dans F est ≥ 2 ;
2. sinon, u est un point d'articulation si F contient un arc (u, v) avec ancêtre[v] $\geq$ début[u] ; cela signifie donc que v ne permet pas de remonter "au-dessus" de u.

L'algorithme 1 calcule et renvoie les numérotations qui nous intéressent, ainsi que les relations de parenté de la forêt d'exploration correspondante.

Algorithme 1 : NumérotationsRécursives(G)

Entrées : un graphe non-orienté G.
Sortie : les numérotations début et ancêtre des sommets dans une forêt d'exploration en profondeur pour G, ainsi que les relations de parenté correspondantes.

```
1  début ← tableau(G.nombre_sommets(), 0);
2  parent ← tableau(G.nombre_sommets(), NIL);
3  ancêtre ← tableau(G.nombre_sommets(), +∞);
4  instant ← 0;
5  Procédure NumérotationRécursive(s)
6  |  instant ← instant + 1;
7  |  début[s] ← ancêtre[s] ← instant;
8  |  pour chaque t ∈ G.voisins(s) faire
9  |  |  si début[t] ≠ 0 alors
10 |  |  |  si parent[s] ≠ t alors
11 |  |  |  |  ancêtre[s] ← min(ancêtre[s], début[t]);
12 |  |  sinon
13 |  |  |  parent[t] ← s;
14 |  |  |  NumérotationRécursive(t);
15 |  |  |  ancêtre[s] ← min(ancêtre[s], ancêtre[t]);
16 pour chaque v ∈ G.sommets() faire
17 |  si début[v] = 0 alors NumérotationRécursive(v) ;
18 renvoyer début, parent, ancêtre
```

L'algorithme 2 utilise les informations calculées par l'algorithme 1 pour en déduire les points d'articulation du graphe.

Algorithme 2 : PointsArticulation(G, début, parent, ancêtre)

Entrées : un graphe non-orienté G et les données de sortie de l'algorithme 1.
Sortie : les points d'articulation de G.

```
1  articulations ← ∅;
   // traitement des racines des arbres d'exploration
2  racines ← {v | v ∈ G.sommets(), parent[v] = NIL};
3  pour chaque départ ∈ racines faire
4  |  si le degré sortant de départ dans la forêt est au moins 2 alors
5  |  |  articulations ← articulations ∪ départ;
   // traitement des autres sommets
6  racines ← racines ∪ {NIL};
7  pour chaque v ∈ G.sommets() faire
8  |  si parent[v] ∉ racines et ancêtre[v] ≥ début[parent[v]] alors
9  |  |  articulations ← articulations ∪ parent[v];
10 renvoyer articulations;
```

Détection des ponts

Les données fournies par l'algorithme 1 sont également utiles pour identifier les ponts dans le graphe : un pont est un arc (u, v) de la forêt d'exploration tel que ancêtre$[v] >$ début$[u]$. Autrement dit, v ne permet pas de remonter "au-dessus" de u, ni à u lui-même. Il nous suffit donc d'examiner les arcs de la forêt, que l'on déduit du tableau parent, et de répertorier les arêtes du graphe qui valident la condition ci-dessus.

Par exemple, à l'étape (2) de l'exemple précédent, les seuls arcs de la forêt vérifiant ce critère sont :
- (c, d), puisque ancêtre$[d] = 4 >$ début$[c] = 3$; et
- (i, l), puisque ancêtre$[l] = 12 >$ début$[i] = 9$.

Les arêtes $\{c, d\}$ et $\{i, l\}$ sont donc bien des ponts, et ce sont les seuls. L'algorithme à écrire se déduit facilement ; on omettra donc son pseudocode.

Renforcement d'un graphe

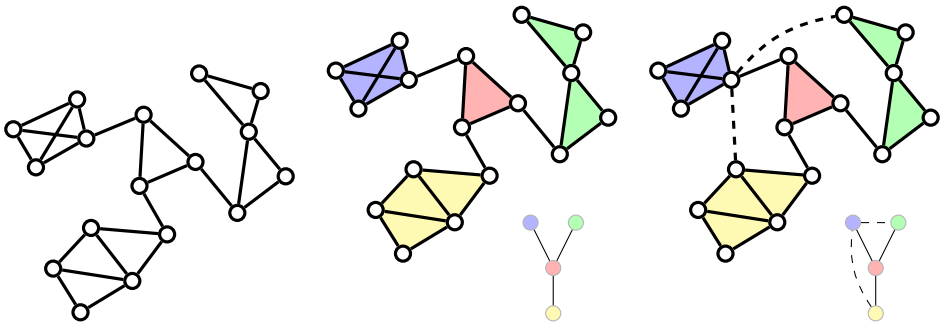
Une fois les faiblesses du graphe identifiées, on peut le renforcer de deux façons qui sont complémentaires : en le modifiant pour qu'aucune arête ne soit un pont, et / ou pour qu'aucun sommet ne soit un point d'articulation.

Ces deux stratégies se concrétisent en rajoutant des arêtes au réseau. Idéalement, on devrait avoir accès au coût de chaque ajout et minimiser le coût total nécessaire au renforcement du réseau, mais ce problème est NP-difficile [2]. On supposera donc que tous les ajouts ont le même coût unitaire, et on tentera donc simplement d'ajouter le moins possible d'arêtes au graphe manipulé. Quelques indications sont données ci-dessous pour obtenir des algorithmes non optimaux mais produisant des solutions de qualité raisonnable.

Élimination des ponts

Les ponts d'un graphe G partitionnent ses sommets en *composantes sans ponts* (abrégées en CSP), qui sont des ensembles maximaux de sommets induisant des sous-graphes sans ponts (mais pas nécessairement sans points d'articulation). On en déduit alors un graphe H dont les sommets sont les CSP de G et dont les arêtes sont les ponts. Si G est connexe, H est forcément un arbre, et on peut se contenter de restreindre nos choix d'arêtes à celles reliant des sommets de G appartenant aux feuilles de H, puisque ce sont les arêtes qui règleront le plus de problèmes.

Cette stratégie est illustrée ci-dessous : (1) on part d'un graphe dont on identifie les CSP à l'étape (2), et trois d'entre elles sont des feuilles, comme le montre l'arbre des CSP accompagnant le graphe. À l'étape (3), on rajoute deux arêtes en pointillés qui nous débarrassent de tous les ponts du graphe. Ici, la solution est optimale, mais ce ne sera pas toujours le cas.



(1) (2) (3)

Seules les feuilles de cet arbre nous intéressent, et on peut les identifier grâce aux ponts. Si $\{u, v\}$ est un pont, alors :

- u et les sommets accessibles au départ de u en évitant v forment une feuille de l'arbre si aucun de ces sommets n'est l'extrémité d'un autre pont ;
- de la même manière, v et les sommets accessibles au départ de v en évitant u forment une feuille de l'arbre si aucun de ces sommets n'est l'extrémité d'un autre pont.

Pour illustrer cela, reprenons l'exemple de la page 2 sur lequel on a déroulé les algorithmes 1 et 2, dont les seuls ponts étaient $\{c, d\}$ et $\{i, l\}$:

- les seuls sommets accessibles au départ de d en évitant c sont $\{d, e, f\}$, et le sous-graphe qu'ils induisent est une CSP qui n'est incidente qu'au pont $\{c, d\}$; dès lors, cette CSP est une feuille du graphe des CSP.
- en revanche, parmi les sommets accessibles au départ de c en évitant d, on trouve i qui est l'extrémité d'un pont ; les sommets $\{a, b, c, g, h, i, j, k\}$ forment donc une CSP, mais cette CSP n'est pas une feuille du graphe des CSP.

Si le graphe G est connexe, on peut éliminer tous ses ponts en reliant les feuilles de H par un chemin arbitraire. Cela signifie donc que l'on sélectionne un sommet arbitraire dans une CSP de G qui est une feuille de H, et qu'on le relie à un autre sommet arbitraire d'une autre CSP de G qui est également une feuille de H, et ainsi de suite jusqu'à ce que les feuilles de H aient été reliées par un chemin.

Si G n'est pas connexe, la même stratégie peut s'appliquer en prenant toutefois garde aux CSP qui sont des sommets isolés dans H : il ne faut pas qu'elles soient des feuilles dans le chemin rajouté, sinon la seule arête rajoutée incidente à une composante isolée sera un pont.

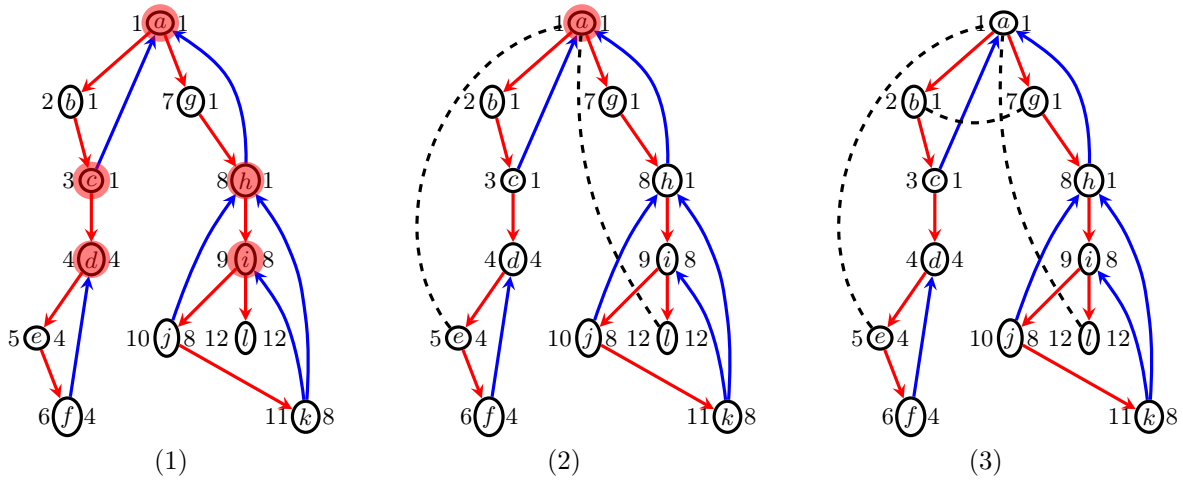
Élimination des points d'articulation

Un réseau sans ponts peut néanmoins contenir des points d'articulation, dont on souhaitera aussi se débarrasser. Pour identifier les arêtes à rajouter au réseau, on peut utiliser une stratégie basée sur la forêt de parcours et les numérotations que l'on a calculées pour obtenir les points d'articulation.

Comme pour l'identification des points d'articulation, on fera une distinction entre les racines de la forêt d'exploration et les autres sommets : si une racine est un point d'articulation, il nous suffit de relier ses descendants par un chemin pour qu'elle ne le soit plus.

Pour les autres sommets, on procède comme suit : comme on l'a vu, si u est un point d'articulation, alors il possède un successeur v dans F à partir duquel on ne peut pas accéder à un ancêtre de u. Dès lors, il nous suffit de rajouter une arête de v vers la racine r de l'arbre de parcours A menant à v pour éliminer tous les points d'articulation situés sur le chemin de r à v dans A. Pour minimiser le nombre d'arêtes à rajouter, il nous faudra donc traiter en premier lieu les points d'articulation les plus "profonds" dans l'exploration, c'est-à-dire ceux dont la date de début est maximale.

Examinons cette stratégie sur l'exemple de la page 2. À l'étape (1), on a reproduit la forêt de parcours et marqué les points d'articulation. Dans l'ordre de parcours en profondeur décroissant, ces points d'articulation sont i, h, d, c, a, que l'on va donc régler dans cet ordre. À l'étape (2), on a trouvé pour i et pour d des descendants (l et e) à relier à la racine a ; en rajoutant les deux arêtes pointillées, on élimine ainsi 4 points d'articulation. L'étape (3) élimine le dernier point d'articulation (a) en reliant ses deux descendants par une arête.



Données

Les jeux de données que vous devez utiliser sont des versions grandement simplifiées de données réelles fournies par la RATP. Elles consistent en un fichier texte par ligne de métro ou de RER. Chacun de ces fichiers possède une section "stations" et une section "connexions" structurées comme suit :

— chaque ligne de la section "stations" suit le format `id:nom`, où `id` est un entier identifiant la station et `nom` est une chaîne contenant le nom habituel de la station (par exemple : "Châtelet-Les Halles").
— chaque ligne de la section "connexions" contient un triplet de la forme `a/b/c`, où `a` et `b` sont les identifiants de deux stations de la ligne concernée et `c` est le temps nécessaire en secondes pour aller de `a` à `b`.

Les identifiants sont réutilisés dans les divers fichiers qui vous sont fournis : on pourra donc retrouver certaines stations sur plusieurs lignes, puisque des correspondances sont possibles entre ces lignes. Pour des raisons d'efficacité, on utilisera les identifiants entiers fournis pour manipuler les sommets, et l'on stockera séparément dans la classe `Graphe` les noms correspondants des stations, que l'on n'utilisera que quand on voudra obtenir un affichage ergonomique.

Remarque : lors de l'exécution du programme, on supposera toujours que les données à charger se trouvent dans le répertoire courant.

Tâches

1. Pour chaque fonction et méthode implémentée, écrivez un test supplémentaire au format doctest. Vous rajouterez tous ces tests dans un seul fichier nommé `mes_tests.txt`, que vous rendrez également.
2. Développez une classe nommée `Graphe` qui vous permettra de stocker les propriétés du réseau qu'il vous faudra construire. Il est fortement conseillé de partir de la classe `DictionnaireAdjacence` qui vous a été fournie. Cette classe `Graphe` devra stocker les informations sur les lignes et les sommets. Attention, il faut adapter la classe à vos données. Lisez bien le sujet, les exemples et les doctests pour comprendre comment faire.

3. Implémentez les fonctions suivantes (consultez les exemples des doctests pour vous aider à comprendre ce qui est attendu) :
   (a) `charger_donnees(graphe, fichier)` : ajoute au graphe les données du fichier dont le nom est passé en paramètre.
   (b) `points_articulation(reseau)` : renvoie les points d'articulation du réseau sous la forme d'un itérable.
   (c) `ponts(reseau)` : renvoie les ponts du réseau sous la forme d'un itérable d'arêtes. N'oubliez pas que deux stations du réseau peuvent être reliés par plusieurs arêtes, et que l'algorithme suggéré n'en tient pas compte : il faudra donc vérifier qu'une arête $\{u, v\}$ identifiée comme un pont est bien la seule arête entre u et v.
   (d) `amelioration_ponts(reseau)` : renvoie un ensemble d'arêtes à rajouter au réseau pour qu'il ne contienne plus de ponts. On n'exige pas une solution optimale, mais on vous demande de tenter de minimiser la taille de cet ensemble. La qualité de la solution fournie interviendra dans la notation.
   (e) `amelioration_points_articulation(reseau)` : renvoie un ensemble d'arêtes à rajouter au réseau pour qu'il ne contienne plus de points d'articulation. Là encore, on n'exige pas une solution optimale, mais on vous demande de tenter de minimiser la taille de cet ensemble. La qualité de la solution fournie interviendra dans la notation.
4. Les options suivantes devront être disponibles dans votre programme. Utilisez le module standard argparse pour implémenter cette partie. La documentation contient de nombreux exemples instructifs.
   — `--metro [lignes]` : précise les lignes de métro que l'on veut charger dans le réseau. Le paramètre `lignes` est facultatif : s'il existe, il s'agit des numéros de lignes qui nous intéressent (par exemple : `--metro 3b 7 14`) ; sinon, on charge toutes les lignes de métro disponibles dans le répertoire courant.
   — `--rer [lignes]` : cf. `--metro`, mais pour les lignes de RER.
   — `--liste-stations` : affiche la liste des stations du réseau avec leur identifiant triées par ordre alphabétique
   — `--articulations` : affiche les points d'articulation du réseau qui a été chargé ;
   — `--ponts` : affiche les ponts du réseau qui a été chargé ;
   — `--ameliorer-articulations` : affiche les points d'articulation du réseau qui a été chargé, ainsi que les arêtes à rajouter pour que ces stations ne soient plus des points d'articulation ;
   — `--ameliorer-ponts` : affiche les ponts du réseau qui a été chargé, ainsi que les arêtes à rajouter pour que ces arêtes ne soient plus des ponts.

Exemples d'exécution

Quelques exemples d'exécution sont donnés ci-dessous. Pour des raisons d'ergonomie, il est nécessaire d'afficher les points d'articulations identifiés dans l'ordre alphabétique. De même, les extrémités de chaque arête affichée seront triées de la même manière, et les listes d'arêtes à affichées seront en outre triée sur la première extrémité.

Afficher toutes les stations d'une ligne

```
$ python3 ameliorations.py --metro 7b --liste-stations
Chargement des lignes ['7b'] de metro ... terminé.
Le réseau contient 8 sommets et 8 arêtes.

Le réseau contient les 8 stations suivantes:

Bolivar (2075)
Botzaris (2002)
Buttes-Chaumont (2013)
```

```
Danube (1635)
Jaurès (1900)
Louis Blanc (1860)
Place des Fêtes (1797)
Pré-Saint-Gervais (1756)
```

Afficher tous les ponts et tous les points d'articulation d'une ligne

```
$ python3 ameliorations.py --metro 7b --ponts --articulations
Chargement des lignes ['7b'] de metro ... terminé.
Le réseau contient 8 sommets et 8 arêtes.

Le réseau contient les 4 ponts suivants:
    - Bolivar -- Buttes-Chaumont
    - Bolivar -- Jaurès
    - Botzaris -- Buttes-Chaumont
    - Jaurès -- Louis Blanc

Le réseau contient les 4 points d'articulation suivants:
   1 : Bolivar
   2 : Botzaris
   3 : Buttes-Chaumont
   4 : Jaurès
```

Afficher les arêtes à ajouter pour éliminer les ponts du réseau

```
$ python3 ameliorations.py --rer  --ameliorer-ponts
Chargement de toutes les lignes de rer ... terminé.
Le réseau contient 92 sommets et 91 arêtes.

On peut éliminer tous les ponts du réseau en rajoutant les 8 arêtes suivantes:
    - Aéroport Charles de Gaulle 2 TGV -- Cergy-Le-Haut
    - Aéroport Charles de Gaulle 2 TGV -- Robinson
    - Boissy-Saint-Léger -- Poissy
    - Boissy-Saint-Léger -- Saint-Germain-en-Laye
    - Cergy-Le-Haut -- Saint-Rémy-lès-Chevreuse
    - Marne-la-Vallée Chessy -- Mitry-Claye
    - Mitry-Claye -- Robinson
    - Poissy -- Saint-Rémy-lès-Chevreuse
```

Modalités

Votre rendu de projet sera constitué des fichiers suivants (noms imposés) :

- `graphe.py` - contiendra votre classe `Graphe` adaptée aux besoins du projet.
- `ameliorations.py` - le programme principal qui devra contenir toutes les fonctions listées dans le sujet.
- les fichiers de données des lignes tels qu'ils vous sont fournis.
- le fichier `mes_tests.txt` contenant les tests demandés plus haut.

Compléments d'information sur Python

Fonctions imbriquées. Python permet de définir des fonctions dans des fonctions. Ceci facilite entre autres la définition des fonctions récursives avec leurs fonctions auxiliaires, et permet de réduire le nombre de paramètres dans les appels récursifs comme dans l'exemple suivant :

```
def trouver_element(liste, x):
    """Renvoie les positions de la liste où x apparaît."""
    positions = set()
    n = len(liste)

    def trouver_element_rec(i=0):
        if i < n:
            if liste[i] == x:
                positions.add(i)
            trouver_element_rec(i+1)

    trouver_element_rec()
    return positions
```

Variables "non-locales". Les fonctions imbriquées ont accès en lecture à toutes les variables de la fonction qui les contient, et en écriture aux variables modifiables. Cependant, les affectations posent problème, car Python traite alors la variable concernée comme une variable locale à la fonction imbriquée :

```
>>> def f():
...     i = 0
...     def g():
...         i = i + 1
...     g()
...     print("après l'appel à g, i =", i)
...
>>> f()
UnboundLocalError: local variable 'i' referenced before assignment
```

Une mauvaise solution consisterait à rendre la variable `i` globale. Au lieu de cela, on utilise le mot-clé `nonlocal` pour expliquer à `g` que `i` n'est pas une variable locale, et qu'il faut donc aller fouiller dans la fonction `f` pour la trouver :

```
>>> def f():
...     i = 0
...     def g():
...         nonlocal i
...         i = i + 1
...     g()
...     print("après l'appel à g, i =", i)
...
>>> f()
après l'appel à g, i = 1
```

Doctests. Les doctests ont été couverts dans des cours précédents, et la documentation correspondante est disponible ici. On peut placer ces doctests dans de simples fichiers texte séparés (`tests.txt`) au lieu de les mettre dans les docstrings des fonctions, et les lancer avec `python3 -m doctest tests.txt`. Si rien ne s'affiche, c'est que tous les tests ont réussi.

Références

[1] S. Even, *Graph Algorithms*, Cambridge University Press, 2ème édition, 2012.

[2] K. P. Eswaran et R. E. Tarjan, *Augmentation problems*, SIAM Journal on Computing, 5 (1976), pages 653–665.