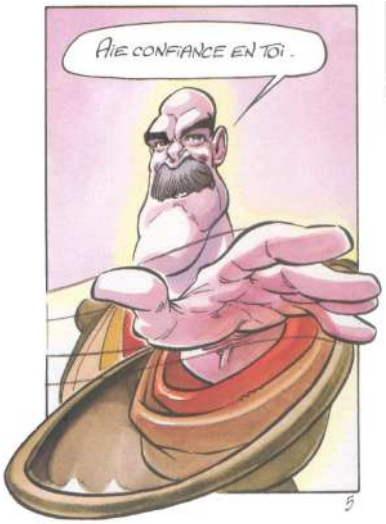
Algorithmique
Partiel n° 3 (P3)

Info-spé - S3#
Epita

12 mai 2021 - 9 : 30

Consignes (à lire) :

- ☐ Vous devez répondre sur **les feuilles de réponses prévues à cet effet**.
  - — Aucune autre feuille ne sera ramassée (gardez vos brouillons pour vous).
  - — Répondez dans les espaces prévus, **les réponses en dehors ne seront pas corrigées** : utilisez des brouillons !
  - — Ne séparez pas les feuilles à moins de pouvoir les ré-agrafer pour les rendre.
  - — Aucune réponse au crayon de papier ne sera corrigée.
- ☐ La présentation est notée en moins, c'est à dire que vous êtes noté sur 20 et que les points de présentation (2 au maximum) sont retirés de cette note.
- ☐ **Le code :**
  - — Tout code doit être écrit dans le langage `Python` (pas de C, Caml, Algo ou autre).
  - — **Tout code `Python` non indenté ne sera pas corrigé.**
  - — Tout ce dont vous avez besoin (classes, fonctions, méthodes) est indiqué dans l'énoncé !
  - — Vous pouvez également écrire vos propres fonctions, dans ce cas **elles doivent être documentées** (on doit savoir ce qu'elles font).
    Dans tous les cas, la dernière fonction écrite doit être celle qui répond à la question.
- ☐ Durée : 2h00



Exercice 1 (Warshall - Trouver-Réunir – *4 points*)

Soit le graphe non orienté $G_1 = < S, A >$, où les sommets sont numérotés de 0 à 8.

Les algorithmes `trouver` et `réunir` (versions non optimisées) appliqués à la liste des arêtes A de G_1 ont permis de construire le vecteur P suivant.

	0	1	2	3	4	5	6	7	8
P	8	5	-1	-1	6	-1	2	3	1

1. Quelles sont les composantes connexes (ensembles de sommets) du graphe G_1 ?

2. Donner la matrice d'adjacence de la fermeture transitive de G_1 (pas de valeur = *faux*, 1 = *vrai*)

3. On applique cette fois-ci les versions optimisées de trouver : avec compression des chemins et réunir : union pondérée à la liste des arêtes A de G_1. Parmi les vecteurs suivants, lesquels pourraient correspondre au résultat ?

P_1	0	1	2	3	4	5	6	7	8
	8	8	4	7	-3	8	4	-2	-4

P_2	0	1	2	3	4	5	6	7	8
	1	-4	6	7	6	1	-3	-2	2

P_3	0	1	2	3	4	5	6	7	8
	8	8	6	7	6	8	-3	-3	-4

P_4	0	1	2	3	4	5	6	7	8
	-4	0	4	7	-3	0	4	-2	0

Exercice 2 (Get Back – *4 points*)

Dans certains problèmes, on utilise un vecteur de marques dans lequel chaque sommet peut avoir *3 valeurs* lors d'un parcours :

— Une valeur (`None` par exemple) pour les sommets non marqués
— Une valeur pour la première rencontre (par exemple 1)
— Une valeur pour la dernière rencontre (par exemple 2)

En utilisant **obligatoirement un vecteur de marques *à 3 valeurs*, peu importe lesquelles** (sans autre vecteur), écrire la fonction `acyclic(G)` qui vérifie si le graphe orienté G est acyclique (sans circuit).

Exercice 3 (Density – *6 points*)

Un graphe non orienté simple (sans liaisons multiples ni boucles) est dit *dense* lorsque le nombre d'arêtes (p) est important par rapport au nombre de sommets (n).

Pour cet exercice on définit la *densité* d'un graphe par la mesure p/n.

1. Pour un **graphe simple connexe** :
   (a) **Le moins dense** : Donner la valeur minimale de p en fonction de n. Quel type de graphe peut avoir ces mesures ?
   (b) **Le plus dense** : Donner la valeur maximale de p en fonction de n. Quel type de graphe peut avoir ces mesures ?
2. Écrire la fonction `density_components` qui retourne la liste des *densités* des composantes connexes d'un graphe non orienté simple.

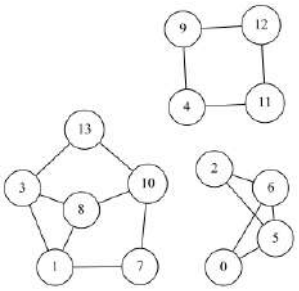


FIGURE 1 – Graphe `G_3cc`

Exemple d'application, avec `G_3cc` le graphe de la figure 1 :

```
>>> density_components(G_3cc)
[1.25, 1.3333333333333333, 1.0]
```

— La première composante a 4 sommets, 5 arêtes
— La deuxième composante a 6 sommets, 8 arêtes
— La troisième composante a 4 sommets, 4 arêtes

Exercice 4 (Levels – *6 points*)

Rappel :

— La **distance** ($distance(x, y)$) entre deux sommets x et y d'un graphe est le nombre d'arêtes d'une **plus courte chaîne** entre ces deux sommets.
— L'**excentricité** d'un sommet x dans un graphe $G =< S, A >$ est définie par :

$$\text{exc}(x) = \max_{y \in S}\{distance(x, y)\}$$

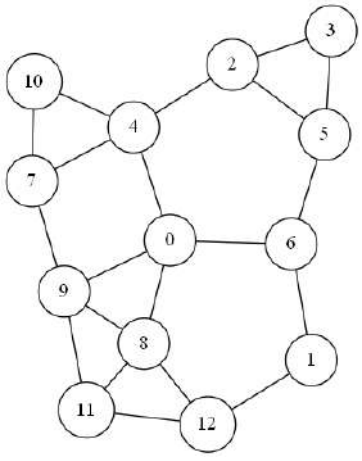


FIGURE 2 – `Gc`

Écrire la fonction `levels(G, src)` qui retourne une liste L de longueur $exc(src) + 1$ dans laquelle chaque valeur $L[i]$ contient les sommets à une distance i de src.

Exemple d'application sur le graphe `Gc` de la figure 2 (l'ordre dans les sous-listes n'est pas important) :

```
>>> levels(Gc, 0)
[[0], [4, 6, 8, 9], [2, 7, 10, 1, 5, 11, 12], [3]]
```

Annexes

Les classes `Graph` et `Queue` sont supposées importées.

Les graphes

Tous les exercices utilisent l'implémentation par listes d'adjacences des graphes.

Les graphes manipulés ne peuvent pas être vides. Il n'y a pas de liaisons multiples ni boucles.

```
class Graph:
    def __init__(self, order, directed = False):
        self.order = order
        self.directed = directed
        self.adjlists = []
        for i in range(order):
            self.adjlists.append([])

    def addedge(self, src, dst):
        self.adjlists[src].append(dst)
        if not self.directed and dst != src:
            self.adjlists[dst].append(src)
```

Les files

- `Queue()` returns a new queue
- q`.enqueue(`e`)` enqueues e in q
- q`.dequeue()` returns the first element of q, dequeued
- q`.isempty()` tests whether q is empty

Autres

- `range`
- `min, max`
- sur les listes :
  - `len(L)`
  - `L.append(elt)`
  - `L.pop()`
  - `L.pop(index)`
  - `L.insert(index, elt)`

  Et n'importe quel opérateur...

Vos fonctions

Vous pouvez également écrire vos propres fonctions, dans ce cas elles doivent être **documentées** (on doit savoir ce qu'elles font).

Dans tous les cas, la dernière fonction écrite doit être celle qui répond à la question.