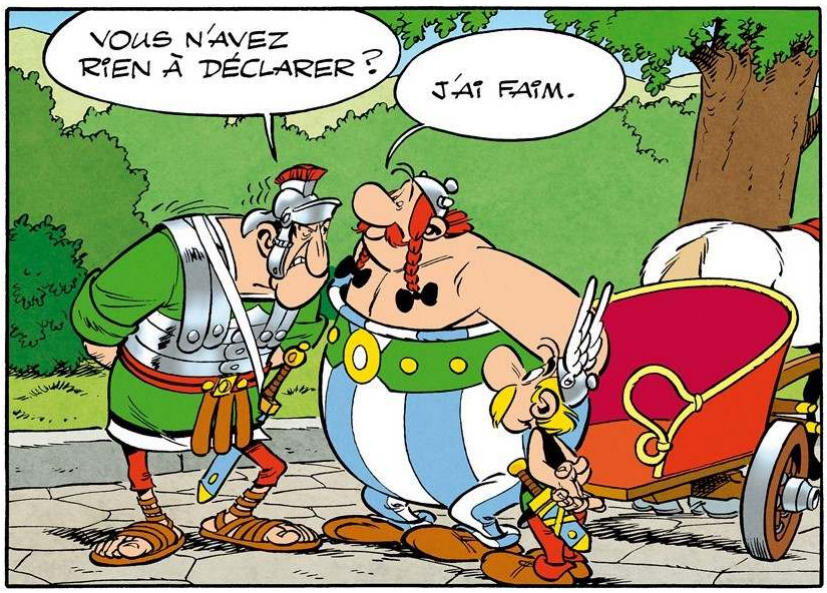
Algorithmique
Partiel n° 3 (P3)

Info-spé - S3#
Epita

14 mai 2019

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 (Graphes : dessiner c'est gagner – 2 *points*)

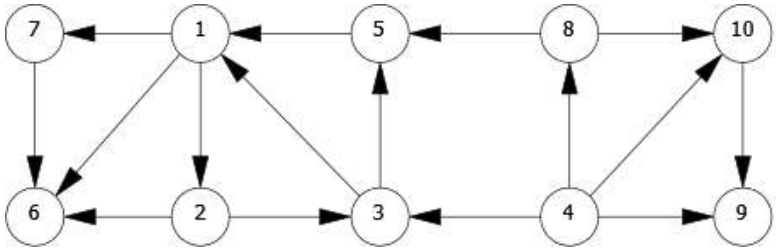


FIGURE 1 – Graphe G_1

Représenter (dessiner) la forêt couvrante associée au parcours en profondeur du graphe G_1 (figure 1). *Ajouter aussi les autres arcs en les qualifiant à l'aide d'une légende explicite.* On considérera le sommet 1 comme base du parcours, les sommets devant être choisis en ordre numérique croissant.

Exercice 2 (Union-Find – 3 *points*)

Soit le graphe non orienté $G = < S, A >$, où les sommets sont numérotés de 0 à 13. Les algorithmes vus en cours `trouver` (sans compression) et `réunir` (version optimisée : union pondérée) ont permis de construire, à partir de la liste des arêtes (A), le vecteur p suivant :

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
p	5	8	5	8	11	-4	5	10	-6	12	8	12	-4	8

1. Donner le nombre de sommets de chaque composante connexe de G (peu importe l'ordre).

2. Quelles arêtes suffit-il d'ajouter pour rendre le graphe connexe ?

3. Parmi les chaînes suivantes, quelles sont celles qui ne peuvent pas exister dans G ?
   — $3 \rightsquigarrow 7$
   — $11 \rightsquigarrow 6$
   — $0 \rightsquigarrow 13$
   — $4 \rightsquigarrow 9$

Exercice 3 (Graphes bipartis (Bipartite graph) – *5 points*)

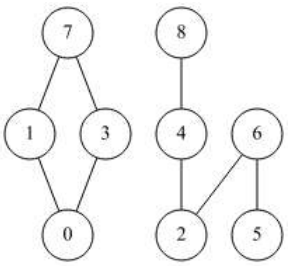


FIGURE 2 – Graphe G_2, biparti

Un graphe biparti est un graphe (multigraphe) non orienté $G = < S, A >$, dans lequel S peut être partitionné en deux ensembles S_1 et S_2 tels que $(u, v) \in A$ implique soit que $u \in S_1$ et $v \in S_2$, soit que $u \in S_2$ et $v \in S_1$. Aucune arête ne doit relier deux sommets d'un même ensemble.

Le graphe G_2 de la figure 2 est biparti avec, par exemple, $S_1 = \{0, 2, 5, 7, 8\}$ et $S_2 = \{1, 3, 4, 6\}$

Écrire une fonction qui teste si un graphe non orienté est biparti.

Exercice 4 (Mangez des crêpes – *8 points*)

Ci-dessous la recette de la crêpe à la banane flambée, avec pour chaque tâche sa durée en secondes.

La recette	Durée (en sec.)	Réf.
Mettre la farine dans un saladier,	3	A
ajouter deux œufs,	30	B
ajouter le lait doucement et mélanger.	600	C
Dans une poêle mettre du rhum.	3	D
Couper les bananes en fines lamelles,	300	E
les mélanger au rhum.	30	F
Faire chauffer le mélange,	120	G
faire flamber le mélange.	10	H
Faire cuire une crêpe,	10	I
verser du mélange bananes-rhum sur la crêpe.	10	J

Quelques précisions concernant l'ordre des tâches :

— la préparation de la pâte à crêpe et celle du mélange rhum-banane peuvent se faire en parallèle ;

— ce n'est que lorsque la crêpe sera cuite et que le mélange sera prêt qu'on pourra verser du mélange sur la crêpe et enfin la déguster ;

— les autres étapes se réalisent séquentiellement : on doit mettre la farine avant les œufs, on doit mettre le rhum avant les bananes (qui peuvent être coupées en avance) dans la poêle.

1. Modéliser la recette sous forme de graphe :
   — Les sommets sont les tâches.
   — Les tâches *start* et *end* représentent le début (commande de la crêpe) et la fin (dégustation) du projet.

   Le graphe qui représente la recette est sans circuit. De plus, tous les sommets sont atteignables depuis un sommet donné (ici la commande de la crêpe = la tâche *start*, sommet n° 0 dans la représentation machine). Dans toute la suite de l'exercice, le graphe aura ces spécifications !

2. **Le cuisinier est tout seul en cuisine :** la durée minimum est donc la somme des temps nécessaires à chaque tâche. Par contre, il faut l'aider à trouver l'ordre dans lequel il va pouvoir faire les différentes tâches : la solution est un tri topologique du graphe.

   (a) Une solution de tri topologique évidente est donnée par l'ordre de la recette. Donner une autre solution : compléter celle donnée sur les feuilles de réponses.

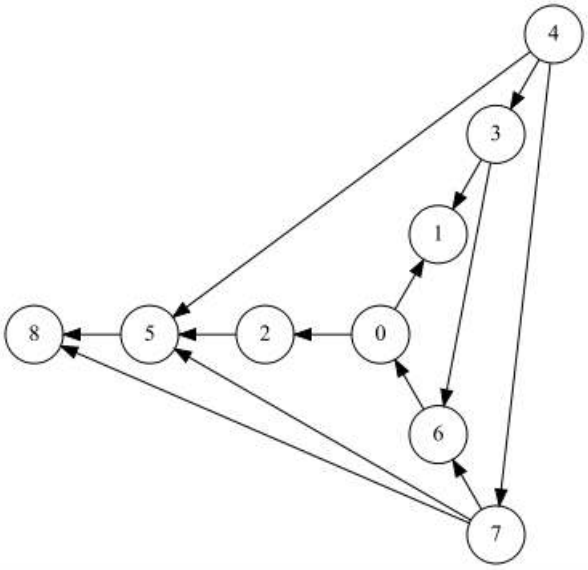
   (b) **Rappel :** Lors d'un parcours profondeur d'un graphe sans circuit, classer l'ensemble des sommets en ordre suffixe inverse est une solution de tri topologique.
   Écrire la fonction qui retourne une solution de tri topologique (une liste de sommets) pour un graphe ayant les mêmes spécifications que ci-dessus.

   (c) Soit une liste de sommets. Écrire une fonction qui vérifie si cette liste peut-être une solution de tri topologique pour un graphe orienté donné (supposé sans circuit).

Exercice 5 (What does it do ? – *4 points*)

Soient les fonctions suivantes :

```
1  def build(G):
2      V = [0] * G.order
3      for x in range(G.order):
4          for y in G.adjlists[x]:
5              V[y] += 1
6      return V
7
8  def what(G):
9      V = build(G)
10     L = []
11     while len(L) < G.order:
12         x = 0
13         while x < G.order and V[x] != 0:
14             x += 1
15         V[x] = -1
16         L.append(x)
17         for y in G.adjlists[x]:
18             V[y] -= 1
19     return L
```



Graphe G_3

1. Donner le résultat de l'application `build(`G_3`)`, avec G_3 le graphe de la figure 5.
2. La fonction `what` :
   (a) Donner le résultat de l'application de la fonction `what(`G_3`)`, avec G_3 le graphe de la figure 5.
   (b) Que représente la liste résultat pour le graphe en paramètre ?
   (c) Quelle doit être la propriété du graphe pour que cette fonction ne "plante" pas ?

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.

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

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

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.