Algorithmique
Connexités
SPÉ S4 EPITA

Examen B7
3 mars 2026

Consignes (à lire) :

- Vous devez répondre sur **les feuilles de réponses prévues à cet effet.**
  Indiquez de manière lisible vos NOM (en majuscule), prénom, UID et classe.
  - Répondez dans les espaces prévus, **les réponses en dehors ne seront pas corrigées.**
  - Aucune réponse au crayon de papier ne sera corrigée.
- La présentation est notée en moins, c'est à dire que vous êtes notés sur 20 et que les points de présentation (2 au maximum) sont retirés de cette note.
- **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.**
  - Les seules classes, fonctions, méthodes que vous pouvez utiliser sont données en **annexe.**
  - Vos fonctions doivent impérativement respecter les exemples d'applications donnés.
  - 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.
  - Comme d'habitude l'optimisation est notée. Si vous écrivez des fonctions non optimisées, vous serez notés sur moins de points. [1]
- Durée : 2h00

Conseil : Commencer par lire le sujet en entier, sans oublier les annexes.

Exercice 1 (L'union fait la force – *3.5 points*)

Rappel : Un graphe G non orienté d'ordre n qui est un *arbre* a trois propriétes :
- G est connexe
- G est acyclique
- G contient $n-1$ arêtes

Soit un graphe G non orienté représenté par n son nombre de sommets et L sa liste d'arêtes (liste de couples de sommets). On désire savoir si le graphe G est un arbre, **sans construire le graphe.**

Écrire la fonction `istreeUF(`n`,` L`)` qui vérifie si le graphe représenté par n et L est un arbre.

Exercice 2 (Big component – *7 points*)

Écrire la fonction `largest_SCC(`G`)` qui retourne la taille (en nombre de sommets) de la plus grande composante fortement connexe du graphe orienté G.

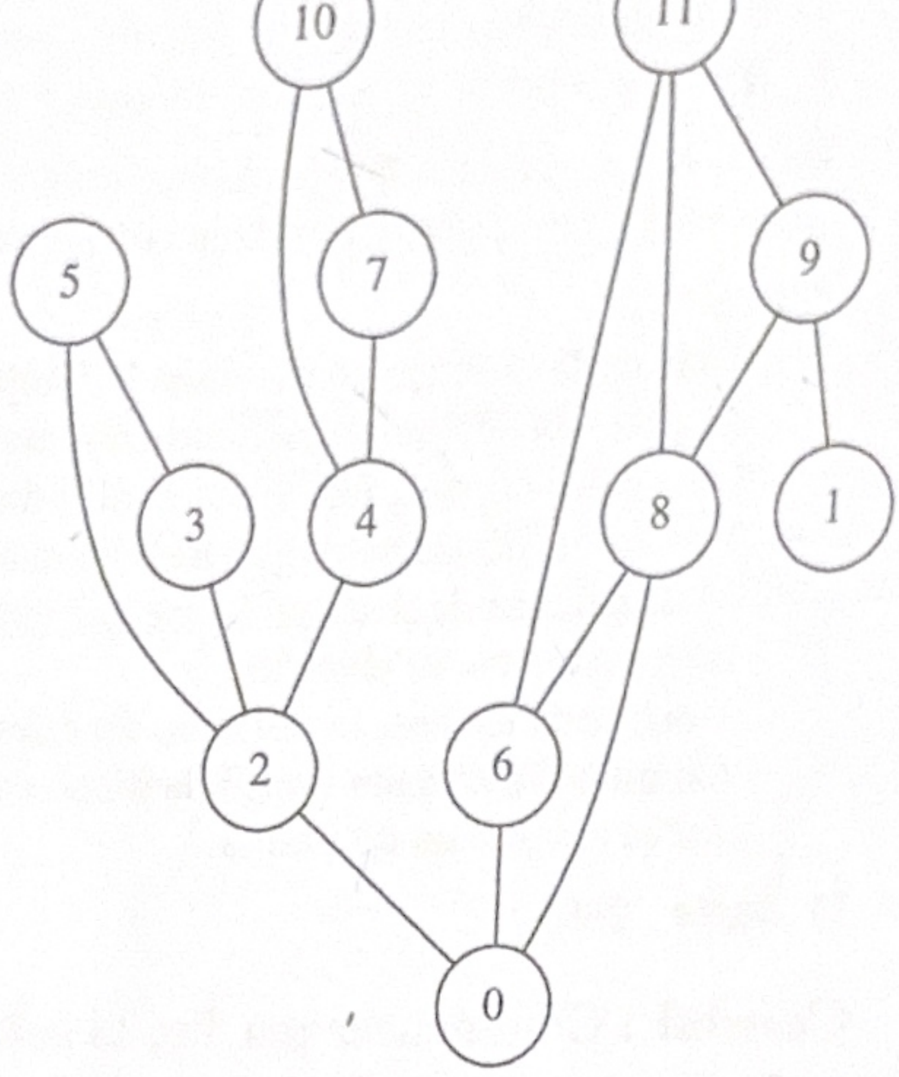
Important : Normalement une page suffit pour la réponse. Si cela est absolument nécessaire, et uniquement dans ce cas, la page suivante peut être utilisée pour ajouter une ou deux fonctions. Dans ce cas, il ne faut surtout pas "couper" une fonction sur deux pages !

---

1. Des fois, il vaut mieux moins de points que pas de points.

Exercice 3 (What is this ? – *5 points*)

```
1  def __what(G, x, D, C):
2      nb = 0
3      p = D[x]
4      for y in G.adjlists[x]:
5          if D[y] == None:
6              D[y] = D[x] + 1
7              (r, n) = __what(G, y, D, C)
8              nb += n
9              p = min(p, r)
10             if r >= D[x] and not C[x]:
11                 C[x] = True
12                 nb += 1
13         else:
14             if D[x] - 1 > D[y]:
15                 p = min(p, D[y])
16     return (p, nb)
17
18
19 def mystery(G):
20     D = [None] * G.order
21     C = [False] * G.order
22     nb = 0
23     c = 0
24     D[0] = 0
25     for y in G.adjlists[0]:
26         if D[y] == None:
27             c += 1
28             D[y] = 1
29             (_, n) = __what(G, y, D, C)
30             nb += n
31             if nb > 2:
32                 return False
33     if c > 1:
34         nb += 1
35         C[0] = True
36     print(D, C)
37     return nb <= 2
```



Graphe *Gmyst*

1. Lors de l'appel `mystery(`*Gmyst*`)` (*Gmyst* le graphe ci-dessus) :
   (a) Donner le contenu du vecteur `D` affiché ligne 36.
   (b) Donner le contenu du vecteur `C` affiché ligne 36 (n'indiquer que les valeurs `True`).
   (c) Quel est le résultat retourné par `mystery(`*Gmyst*`)` ?
2. Soit G un graphe connexe, sur lequel est appliqué `mystery` :
   (a) Pour un sommet `s`, que représente `D[s]` ?
   (b) Pour un sommet `s`, que représente `C[s]` ?
   (c) Que teste la fonction `mystery` ?

Exercice 4 (Warshall, Connect Me – *4.5 points*)

Rappels :

- On appelle *fermeture transitive* d'un graphe non orienté G défini par le couple $< S, A >$, le graphe G^* défini par le couple $< S, A^* >$ tel que pour toutes paires de sommets $x, y \in S$, il existe une arête $\{x, y\}$ dans G^* si-et-seulement-si il existe une chaîne entre x et y dans G.
- L'algorithme de Warshall calcule la matrice d'adjacence de la fermeture transitive d'un graphe.

Écrire la fonction `connect_me(M)` qui, à partir de M la matrice résultat de l'algorithme de Warshall appliquée au graphe non orienté G, retourne la liste des arêtes (une liste de couples) à ajouter à G pour le rendre connexe. Moins il y a d'arêtes, mieux c'est.

Exemples d'applications

Matrices d'adjacence de fermetures transitives dans lesquelles 1 = `True` et "vide" = `False` :

	0	1	2	3	4	5	6	7	8	9
0	1		1			1	1			
1		1		1				1	1	
2	1		1			1	1			
3		1		1				1	1	
4					1					1
5	1		1			1	1			
6	1		1			1	1			
7		1		1				1	1	
8		1		1				1	1	
9					1					1

Matrice M2

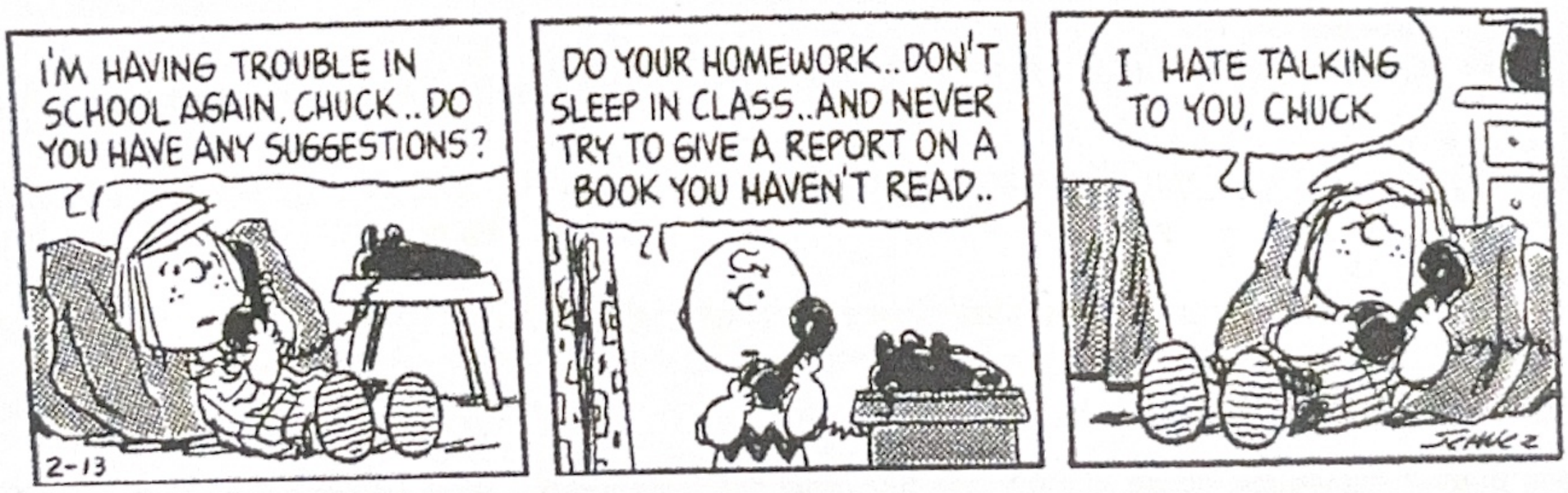
	0	1	2	3	4	5	6	7	8	9
0	1	1				1				
1	1	1				1				
2			1		1		1			
3				1				1		
4			1		1		1			
5	1	1				1				
6			1		1		1			
7				1				1		
8									1	1
9									1	1

Matrice M3

Résultats possibles sur les matrices données (il y a plusieurs résultats corrects) :

```
>>> connect_me(M2)
[(0, 1), (0, 4)]    # [(0, 1), (1, 4)]

>>> connect_me(M3)
[(0, 2), (0, 3), (0, 8)]    # [(0, 2), (2, 3), (3, 8)]
```



Annexes

Les graphes

Tous les exercices utilisent l'implémentation par listes d'adjacence des graphes. Les listes d'adjacence sont triées en ordre croissant.

Les graphes manipulés ne peuvent pas être vides (ni l'ensemble des sommets, ni celui des liaisons ne sont vides). Il n'y a pas de liaisons multiples ni boucles. Il n'y a pas de sommets isolés.

Rappel des attributs utiles des graphes :

```
# G: Graph
G.order
G.adjlists # G.adjlists[x] contains the adjacent vertices of x
```

Les piles

- `Stack()` retourne une nouvelle pile
- S`.push(`e`)` empile e dans S
- S`.peek()` retourne l'élément au sommet de S
- S`.pop()` retourne l'élément au sommet de S, dépilé
- S`.isempty()` teste si S est vide

Autres

- sur les listes : `len`, `append`, `pop`
- `range`
- `min`, `max`, `abs`

Fonctions données

```
def find(x, P):
    rx = x
    while P[rx] >= 0:
        rx = P[rx]
    return rx

def union(x, y, P):
    rx = find(x, P)
    ry = find(y, P)
    if rx != ry:
        if P[rx] < P[ry]:
            P[rx] = P[rx] + P[ry]
            P[ry] = rx
        else:
            P[ry] = P[ry] + P[rx]
            P[rx] = ry
        return True
    else:
        return False

def build(L, n):
    '''
    n: integer > 0
    L: list of pairs (a, b) with a and b in [0, n[
    '''
    P = [-1] * n
    for (x, y) in L:
        union(x, y, P)
    return P
```

Vos fonctions

Vous pouvez également écrire vos propres fonctions supplémentaires, dans ce cas vous devez donner leurs **spécifications** (on doit savoir ce qu'elles font).

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