Exercice 1. Composantes fortement connexes

Question 1 –

```
MFig1 = [[0,1,1,1,0],
         [0,0,0,0,0],
         [0,1,0,1,0],
         [1,0,0,0,0],
         [0,1,0,0,1]]
```

Question 2 – Les composantes fortement connexes de ce graphe sont :

$$S_1 = \{0, 1, 3\}$$
$$S_2 = \{2\}$$
$$S_3 = \{4, 5, 6, 7, 11, 12\}$$
$$S_4 = \{8, 9, 10\}$$

Question 3 – On utilise l'algorithme de parcours en largeur vu en cours.

```
import collections
def accessibles(M, u0):
    n = len(M)
    B = {u: True for u in range(n)} # Sommets blancs
    B[u0] = False
    G = collections.deque([u0])     # Sommets gris
    N = {}                          # Sommets noirs
    while len(G) > 0:
        u = G.popleft()
        N[u] = None
        for v in range(n):
            if M[u][v] == 1 and B[v]:
                G.append(v)
                B[v] = False
    return N
```

Question 4.a –

```
def CFC(Acc, u0):
    L = []
    for u in Acc[u0]:
        if u0 in Acc[u]:
            L.append(u)
    return L
```

Question 4.b –

```
def ListeCFC(M):
    n = len(M)
    Acc = [accessibles(M, u) for u in range(n)]
    M = []
    dans_M = [False] * n
    for u in range(n):
        if not dans_M[u]:
            L = CFC(Acc, u)
            for v in L:
                dans_M[v] = True
            M.append(L)
    return M
```

Exercice 2. Tri topologique

Référence : “Algorithmique” - Cormen, Leiserson, Rivest, Stein.

Question 1.a –

```
def test1(n, T1):
    return len(T1) == n
```

Question 1.b – La fonction est de complexité constante.

Question 2 – Pour obtenir une complexité linéaire, on créé d'abord un dictionnaire dont les clés sont les éléments de `T1`. On vérifie ensuite que tous les éléments de $[\![0\,;\,n-1]\!]$ sont des clés du dictionnaire.

```
def test2(n, T1):
    d1 = {e: None for e in T1}
    for i in range(n):
        if i not in d1:
            return False
    return True
```

Question 3.a –

```
1 def inverse(T1):
2     T2 = [None for _ in range(len(T1))]
3     for i in range(len(T1)):
4         T2[T1[i]] = i
5     return T2
```

Question 3.b –

```
1 def test3(G, T1):
2     T2 = inverse(T1)
3     for u in range(len(G)):
4         for v in G[u]:
5             if T2[u] >= T2[v]:
6                 return False
7     return True
```

Question 3.c – On s'intéresse d'abord à chacune des lignes de la fonction `inverse` :

- La création de la liste (ligne 2) prend un temps $\Theta(n)$.
- La boucle `for` fait n tours et chaque tour s'exécute en temps constant.
- Mise à part la ligne 2, toutes les lignes s'exécutent en temps constant.

Ainsi, le temps d'exécution de la fonction `inverse` est en $\Theta(n)$.

On s'intéresse maintenant au temps d'exécution de la fonction `test3`. Si on note n le nombre de sommets de G et m son nombre d'arcs, alors :

- La ligne 2 s'exécute en temps $\Theta(n)$.
- La boucle de la ligne 3 fait exactement n tours.
- Pour un sommet `u` fixé, la boucle de la ligne 4 parcourt tous les successeurs de `u`. Ainsi, au maximum, la ligne 5 est exécutée une fois pour chaque arc.
- À part la ligne 2, toutes les lignes s'exécutent en temps constant.

Au total, le temps d'exécution de la fonction `test3` est en

$\boxed{\mathcal{O}(n+m) \text{ avec } n \text{ le nombre de sommets et } m \text{ le nombre d'arcs}}$

Question 4 –

```
def estTriTopo(G, T1):
    n = len(G)
    return test1(n, T1) and test2(n, T1) and test3(G, T1)
```

Question 5 –

```
def supprSommets(G, L):
    _, S = G
    for u in L:
        S[u] = False
```

Question 6 –

```
def degEntrant(G):
    G0, S = G
    D = []
    for u in range(len(S)):
        if not S[u]:
            D.append(-1)
        else:
            D.append(0)
            for v in G0[u]:
                if S[v]:
                    D[v] += 1
    return D
```

Question 7 – Supposons par l'absurde disposer d'un graphe G contenant un circuit C ainsi que d'un tri topologique T. Le circuit C est de la forme $(u_0, u_1, \ldots, u_k)$ où $k \geqslant 1$, $u_0 = u_k$ et (u_i, u_{i+1}) est un arc pour tout i. Si on note ℓ_i la position de u_i dans T, on obtient :

$$\ell_0 < \ell_1 < \ldots < \ell_k.$$

Or, $u_0 = u_k$, donc $\ell_0 = \ell_k$ et donc l'inégalité $\ell_0 < \ell_k$ est absurde.

Question 8 – Supposons par l'absurde que dans G tous les sommets aient un degré entrant strictement positif. Soit u_0 un sommet quelconque de G (un tel sommet existe par hypothèse). Le degré entrant de u_0 est strictement positif, donc il existe un sommet u_1 tel que (u_1, u_0) est un arc dans G. De même, il existe un sommet u_2 tel que (u_2, u_1) est un arc dans G. Plus généralement, en utilisant une récurrence, il est possible de construire une suite infinie de sommets $(u_0, u_1, u_2, \ldots)$ telle que pour tout i, (u_{i+1}, u_i) est un arc de G. Comme le nombre de sommets dans G est fini, il y a forcément des doublons dans cette liste infinie de sommets. On peut donc définir k le plus petit entier tel que u_k apparaît déjà dans $(u_0, u_1, \ldots, u_{k-1})$ et ℓ l'unique entier tel que $\ell < k$ et $u_\ell = u_k$.

Le chemin $(u_k, u_{k-1}, u_{k-2}, \ldots, u_{\ell+1}, u_\ell)$ est alors un circuit de G et donc de G_0, ce qui contredit l'hypothèse de l'énoncé.

Question 9 –

```
def triTopo1(G0):
    n = len(G0)
    G = (G0, [True]*n)
    T = []
    while len(T) < n:
        D = degEntrant(G)
        L = [u for u in range(n) if D[u] == 0]
        T = T+L
        supprSommets(G, L)
    return T
```

Question 10.a –

	Étape 5	Étape 6	Étape 7
T	[0,1,2,6]	[0,1,2,6,4]	[0,1,2,6,4,5]
D	[0,0,0,1,0,0,0,2,1]	[0,0,0,1,0,0,0,1,1]	[0,0,0,0,0,0,0,1,0]
F	(4,5)	(5)	(3,8)

Question 10.b – À la fin de la procédure, on obtient :

[0,1,2,6,4,5,3,8,7]

Question 11 –

```
import collections
def triTopo2(G):
    n = len(G)
    T = []
    D = degEntrant((G, [True]*n))
    F = collections.deque([u for u in range(n) if D[u] == 0])
    while len(F) > 0:
        u = F.popleft()
        T.append(u)
        for v in G[u]:
            D[v] -= 1
            if D[v] == 0:
                F.append(v)
    return T
```

Question 12.a – Par l'absurde, supposons qu'il existe un arc (u, v) tel que $deb(v) < deb(u)$ et $fin(v) > fin(u)$. L'appel à $PP(v)$ se décompose en trois étapes :

(i) Début de traitement du sommet v. En d'autres termes, on fixe la valeur de $deb(v)$.

(ii) On lance les appels récursifs.

(iii) Fin de traitement du sommet v. En d'autres termes, on fixe la valeur de $fin(v)$.

L'appel à $PP(u)$ a pu se produire à trois moments différents :

- Avant l'étape (i), mais dans ce cas on aurait $deb(u) < deb(v)$ ce qui contredit notre hypothèse.
- Après l'étape (iii), mais dans ce cas on aurait $fin(u) > fin(v)$ ce qui contredit notre hypothèse.
- Pendant l'étape (ii).

Ainsi, la seule possibilité est que l'appel à $PP(u)$ se soit produit pendant l'étape (ii). Lors des appels récursifs, la fonction PP n'emprunte que des arcs du graphe pour se déplacer entre les sommets. Il existe donc un chemin entre v et u dans G. Ce chemin peut être complété en un circuit en utilisant l'arc (u, v), ce qui constitue une contradiction.

Question 12.b – Montrons que l'ordre postfixe lu à l'envers est un tri topologique (dans l'exemple de l'énoncé, il s'agit de montrer que (6,2,5,3,1,0,8,4,7) est un tri topologique). On doit montrer que pour tout arc (u, v) dans le graphe, u apparaît après v dans l'ordre postfixe, c'est à dire que $fin(u) > fin(v)$. Pour le montrer, on distingue 2 cas :

- ⋆ Si $deb(v) < deb(u)$ alors d'après la question précédente, on a $fin(v) < fin(u)$.
- ⋆ Si $deb(v) > deb(u)$ alors au début de l'appel à $PP(u)$, le sommet v n'a pas encore été exploré.

Puisqu'il existe un arc entre u et v, on en déduit que $PP(v)$ est l'un des appels récursifs de $PP(u)$. On a donc $fin(v) < fin(u)$.

Question 13 –

- La liste « `R: list[bool]` » est telle que `R[u]` vaut `True` si et seulement si le sommet `u` a déjà été rencontré lors d'un parcours.
- La liste « `post: list[int]` » est la liste des sommets dans l'ordre postfixe.

```
# Procédure PP décrite dans l'énoncé
def PP(G, u, R, post):
    R[u] = True
    for v in G[u]:
        if not R[v]:
            PP(G, v, R, post)
    post.append(u)
```

```
def triTopo3(G):
    n = len(G)
    R = [False] * n
    post = []
    for u in range(n):
        if not R[u]:
            PP(G, u, R, post)
    return post[::-1]
```

Question 14.a –

```
def perm(L):
    if len(L) == 0:
        return [[]]
    P = []
    for i in range(len(L)-1, -1, -1):
        P0 = perm(L[:i] + L[i+1:])
        for M in P0:
            M.append(L[i])
            P.append(M)
    return P
```

Question 14.b –

```
def compterTT(G):
    n = len(G)
    L = perm([i for i in range(n)])
    L = [T for T in L if estTriTopo(G, T)]
    # return L
    return len(L)
```