DS d'informatique n°5 – Corrigé

/ 3 **1)** `Gfig2 = [ [4,8], [], [3,5], [], [7], [3,8], [3], [], [7] ]`

/ 1,25 **2) a)**

```
1 def test1(n,T):
2     if len(T)==n:
3         return True
4     else:
5         return False
```

/ 1,75 **b)** Il n'y a qu'une opération élémentaire : == ligne 2 (et éventuellement `len(T)` ligne 2). Il n'y a pas de pire cas : quels que soient les arguments, on effectue une (ou deux) opérations élémentaires. La complexité est donc constante, ou encore d'ordre 1 ou en $O(1)$.

/ 8+7 **3)** Il faut bien comprendre que `T` ne vérifie pas la condition **(C1)** a priori. En particulier, même s'il doit verifier **(C2)**, donc être de taille au moins n, cela ne l'empêche pas de contenir d'autres éléments que des entiers de 0 à $n-1$.

```
1  def test2(n,T):
2      m = len(T) # a priori T peut être de taille différente de n...
3      if m<n:
4          return False # T ne pourra pas vérifier (C2) dans ce cas
5
6      # on vérifie d'abord que T ne contient pas de valeur en double
           parmi les entiers de 0 à n-1
7      Verif = [False for k in range(n)] # Verif[k] vaudra True si k
           est dans T
8      for i in range(m):
9          nombre = T[i]
10         if type(nombre)==int and nombre >= 0 and nombre <= n-1:
11             if Verif[nombre] == False: # on n'a jamais trouvé
                   nombre dans T jusqu'à présent
12                 Verif[nombre] = True
13             else: # on l'a déjà trouvé : (C2) est fausse
14                 return False
15
16     # à ce stade, T1 ne peut pas contenir deux fois un même entier
           de [0,n-1]. Il reste à vérifier qu'il contient tous les
           entiers au moins une fois
17     for k in range(n):
18         if Verif[k] == False:
19             return False # il manque l'entier k dans T
20     return True
```

Justification "succinte" de la complexité linéaire (non nécessaire sur la copie) : la création de `Verif` constitue n opérations élémentaires puis on réalise n itérations de deux boucles (non imbriquées). Chaque itération de la première réalise au maximum 6 (ou 7 selon la convention) opérations élémentaires. La seconde n'en réalise qu'une. Donc la complexité est linéaire.

/5,5+2,5

4) a)

```python
def inverse(T1):
    n = len(T1)
    T2 = [-1]*n   # -1 est arbitraire, ils seront tous écrasés

    for k in range(n):
        u = T1[k]  # par (C1)+(C2), u est un entier de 0 à n-1
        T2[u] = k  # k est l'indice de u dans T1

    # T2 ne contient plus de -1 car au cours de la boucle, u
       prend toutes les valeurs de 0 à n-1, grâce à (C2)
    return T2
```

Justification "succinte" de la complexité linéaire (non nécessaire sur la copie) : la création de `T2` constitue n opérations élémentaires puis on réalise n itérations d'une boucle qui, selon certaines conventions, ne contiendrait pas d'opération élementaire ! Dans les faits, des opérations de lecture d'élément d'une liste (`T1[k]`) ou d'écriture (`T2[u]=...`) peuvent constituer une opération élémentaire. Dans tous les cas, la complexité est linéaire.

/ 10

b)

```python
def test3(G,T1):
    T2 = inverse(T1)
    n = len(G)
    for u in range(n):
        for v in G[u]: # pour chaque successeur v de u dans G
            # il faut vérifier que u apparait avant v dans T1.
               On se sert pour cela de la liste T2

            if not( T2[u]<T2[v] ) # si u n'apparait pas
               strictement avant v dans T1
                return False
    return True
```

/ 3

5)

```python
def estTriTopo(G,T):
    n = len(G)
    if test1(n,T)==True:
        if test2(n,T)==True:
            if test3(G,T)==True: # il faut que test1 et test2
               renvoient True avant de vérifier avec test3
                return True
    return False
```

/ 6

6)

```
def supprSommets(G,L):
    G0 = G[0] # premier élément du couple
    S = G[1]  # deuxième élément du couple
    for sommet in L:
        # il faut retirer sommet du graphe G
        S[sommet]=False # cela revient à faire ceci
    return None
```

/ 12

7)

```
def degEntrant(G):
    G0 = G[0] # premier élément du couple
    S = G[1]  # deuxième élément du couple
    n = len(G0) # nombre de sommets dans le graphe G0
    D = [0]*n

    for w in range(n):
        if S[w]==False: # pour chaque sommet w qui n'est pas dans G
            D[w]=-1

    for u in range(n):
        if S[u]==True: # pour chaque sommet u dans G
            for v in G0[u]:
                if S[v]==True:
                    # on ajoute 1 aux degrés entrants des
                       successeurs de u qui sont dans G
                    D[v] = D[v]+1
                    # dans cette boucle, on n'a pas touché aux
                       valeurs de D[w] fixées à -1 dans la boucle
                       précédente.

    return D
```

/ 8

8) 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 $(u_0, u_1, \cdots, u_k)$ avec $k \geq 1$, $u_0 = u_k$ et la condition que (u_i, u_{i+1}) soit un arc pour tout i. Si on note ℓ_i la position de u_i dans T, par la condition **(C3)**, on obtient :

$$\ell_0 < \ell_1 < \ell_2 < \ldots < \ell_k$$

Or, $u_0 = u_k$ et donc $\ell_0 = \ell_k$. Contradiction.

/ 8

9) Supposons par l'absurde que dans G tous les sommets aient un degré entrant strictement positif. Soit u_0 un sommet quelconque de G. 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 $u_0, u_1, u_2, \cdots$ 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 parmi $u_0, u_1, \cdots, u_{k-1}$ et ℓ l'unique entier tel que $\ell < k$ et $u_\ell = u_k$.

Le chemin $(u_k, u_{k-1}, \cdots, u_{\ell+1}, u_\ell)$ est alors un circuit de G (aucune arête n'est empruntée deux fois puisque les sommets de départ $u_k, u_{k-1}, \cdots, u_{\ell+1}$ sont tous distincts) et donc de G_0, ce qui contredit l'hypothèse de l'énoncé.

/ 12 **10)**

```
def triTopo1(G):
    G0 = G[0]
    n = len(G0) # nombre de sommets de G0
    T = []
    while len(T) < n:
        D = degEntrant(G)
        L = [] # liste des sommets ayant un degré entrant nul
        for u in range(n):
            if D[u]==0:
                L.append(u)
        T = T+L # on ajoute ces sommets au tri
        supprSommets(G, L) # on les enlève de G
    return T
```

/ 4,5 **11) 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	(5,4)	(5)	(8,3)

/ 2 **b)** On obtient le tri topologique

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

/ 15

```
import collections
def triTopo2(G):
    G0 = G[0]
    n = len(G0) # nombre de sommets de G0
    D = degEntrant(G)
    F = collections.deque()
    for u in range(n):
        if D[u]==0:
            F.appendleft(u) # on met dans la file les sommets
                ayant un degré entrant nul
    T = [] # liste du tri topologique
    while len(F) > 0:
        u = F.pop()
        T.append(u) # le sommet u étant enlevé...
        for v in G0[u]: # les degrés entrants des successeurs v de
            u diminuent de 1 (même si v n'est plus dans G, c'est OK)
            D[v] = D[v]-1
            if D[v]==0:  # si l'un d'eux est de degré entrant 0,
                alors on l'ajoute à F
                F.append(v)
    return T
```