

1. Changer de représentation

1. On initialise avec une matrice représentant un graphe sans arête puis on rajoute les arêtes du graphe :

```
def matrice(d) :
    n = len(d)
    M = [[0 for j in range(n)] for i in range(n)]
    for i in d :
        for j in d[i] :
            M[i][j] = 1
    return M
```

2. Même chose dans l'autre sens :

```
def Dico(M) :
    n = len(M)
    d = {i:[] for i in range(n)}
    for i in range(n) :
        for j in range(n) :
            if M[i][j] == 1 :
                d[i].append(j)
    return d
```

2. Orienté ou non

1. On vérifie la présence des arêtes « de retour », jusqu'à trouver une éventuelle absence :

```
def oriente(d) :
    for c in d :
        for v in d[c] :
            if not c in d[v] :
                return True
    return False
```

2. La complexité est en $O(n^3)$ car le test `c in d[v]` (appartenance à une liste) est en $O(n)$.

Pour limiter la complexité, on peut commencer par changer de représentation et passer par la matrice d'adjacence du graphe : la fonction `matrice` est de complexité $O(n^2)$ puis on vérifie si la matrice est symétrique, à nouveau avec une complexité en $O(n^2)$.

3. Composante fortement connexe

1. C'est exactement le code du parcours en profondeur vu en cours (le parcours en largeur pourrait aussi convenir) :

```
def accessible(i,d) :
    p = deque()
    vus = {x:False for x in G}
    Acces = []
    p.append(i)
    while len(p)>0 :
        som = p.pop()
        if not vus[som] :
            vus[som] = True
            Acces.append(som)
        for s in d[som] :
            if not vus[s] :
                p.append(s)
    return Acces
```

2. On part d'un graphe « vide » et on rajoute les arêtes inverses :

```
def transpose(d) :  
    dt = {c:[] for c in d}  
    for c in d :  
        for v in d[c] :  
            dt[v].append(c)  
    return dt
```

1. On cherche les sommets accessibles depuis *i* sur le graphe *d*, puis ceux depuis lesquels on peut accéder à *i* (ce sont les sommets accessibles depuis *i* sur le graphe transposé). La CFC contenant *i* est alors l'intersection des deux listes précédentes. Reste à faire cette recherche pour tous les sommets *i* ; la liste finale comporte plusieurs fois la même CFC.

Une fonction pour l'intersection de deux listes : **set** convertit la liste en ensemble et **&** est le symbole pour l'intersection ; la conversion en ensemble permet de supprimer les doublons au passage. On peut bien sûr programmer une telle fonction directement avec des listes : on initialise une liste *L* vide, on parcourt *L1*, si les éléments sont aussi dans *L2*, on les rajoute dans *L* que l'on renvoie à la fin.

```
def intersection(L1,L2) :  
    return list(set(L1)&set(L2))
```

Puis la fonction CFC

```
def CFC(d) :  
    L = []  
    dt = transpose(d)  
    for c in d :  
        L1 = accessible(c,d)  
        L2 = accessible(c,dt)  
        L.append(intersection(L1,L2))  
    return L
```

4. Détecter un cycle

1. On effectue le parcours jusqu'à retomber sur le sommet *i* ; il ne faut pas faire la comparaison avec **som** car on a **som** = *i* au premier passage dans la boucle **while** :

```
from collections import deque  
  
def cycle(i,d) :  
    p = deque()  
    vus = {x:False for x in G}  
    p.append(i)  
    while len(p)>0 :  
        som = p.pop()  
        if not vus[som] :  
            vus[som] = True  
        for s in d[som] :  
            if s == i :  
                return True  
            if not vus[s] :  
                p.append(s)  
    return False
```

2. On introduit un dictionnaire **origine** qui permet de mémoriser depuis quel sommet on est arrivé au sommet courant.

```
def cycleNO(i,d) :
    p = deque()
    vus = {x:False for x in G}
    p.append(i)
    origine = {}
    while len(p)>0 :
        som = p.pop()
        if not vus[som] :
            vus[som] = True
            for s in d[som] :
                if s == i and origine[som] != i:           # on peut aller en i et ce
                    n'est pas de là que l'on venait
                    return True
            if not vus[s] :
                p.append(s)
                origine[s] = som
    return False
```