I Algorithmes sur les graphes

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 $O(n)$.

3. Détecter un cycle

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
```

4. 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 parcours `L1`, si les éléments sont aussi dans `L2`, on les rajoute dans `L` que l'on revoie 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
```

II Chasse au trésor sur un graphe

1. Le parcours est $1, 2, 4, 6, 4, 2, 1, 3, 5$ donc de longueur 8 alors qu'il y a plus court : $1, 3, 5, 3, 1, 2, 4, 6$ qui est de longueur 7.

2. a) Le parcours en largeur est un parcours par distance croissante donc qui permet de trouver le sommet le plus proche vérifiant une certaine condition.

   b) Dans le code classique du parcours en profondeur, on arrête la recherche dès qu'on a trouvé une nouvelle pièce et on introduit un dictionnaire `origine` pour pouvoir mémoriser le chemin à faire : si `s` est un sommet visité, `origine[s]` est le sommet qui permet d'arriver au sommet `s`

```python
from collections import deque

def suivant(s,G,T) :
    file = deque()
    vus = {c:False for c in G}
    origine = {}
    file.append(s)
    trouve = False
    while not trouve :
        w = file.popleft()
        if not vus[w] :
            vus[w] = True
        for u in G[w] :
            if not vus[u] :
                file.append(u)
                if u not in origine :
                    origine[u] = w
        if T[w] :
            trouve = True
    chemin = [] # on reconstruit le chemin (à l'envers)
    while w != s :
        chemin.append(w)
        w = origine[w]
    return chemin[::-1]  # et on le retourne
```

3. On commence par compter le nombre de pièces pour savoir quand la récolte sera terminée.

```python
def recolte(s,G,T) :
    n = 0
    for c in T :
        if T[c] :
            n += 1
    trajet = [s]
    for k in range(n) :
        L = suivant(s,G,T)
        trajet += L
        s = trajet[-1] # le sommet dont il faut repartir
        T[s] = False # il n'y a plus de pièce sur ce sommet
    return trajet
```