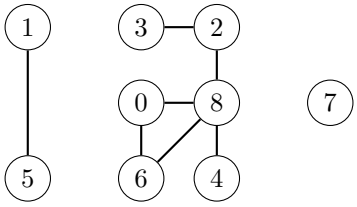
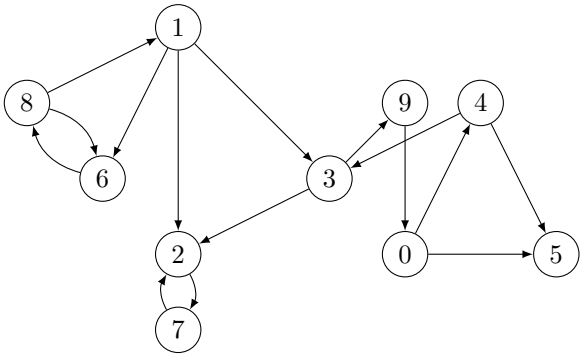
Dans tout le TP, on travaille avec des graphes représentés par tableaux de listes d'adjacence.

```
type graphe = int list array
```

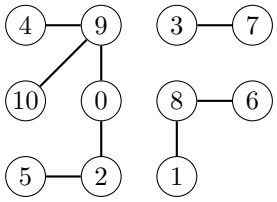
Créer des variables `g1`, `g2`, `g3` et `g4` correspondant aux tableaux de listes d'adjacence des graphes suivants :



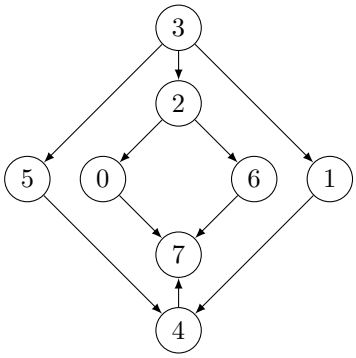
Graphe G_1



Graphe G_2



Graphe G_3



Graphe G_4

Toutes les fonctions de ce TP devront avoir une complexité linéaire en $|S| + |A|$ pour un graphe $G = (S, A)$.

1 Graphes non orientés

Exercice 1

1. Écrire une fonction `composantes_connexes : graphe -> int list list` qui prend en argument un graphe supposé non orienté et renvoie une liste de listes d'entiers, chaque sous-liste correspondant à une composante connexe du graphe donné en argument.
2. Écrire une fonction `acyclique_no : graphe -> bool` qui prend en argument un graphe supposé non orienté et renvoie un booléen qui vaut `true` si et seulement si le graphe ne contient pas de cycle.

   Indication : on pourra utiliser la fonction précédente.

2 Graphes orientés

Exercice 2

1. Écrire une fonction `numeros_DFS : graphe -> int array * int array` qui prend en argument un graphe (orienté ou non) G et renvoie un couple de tableaux `pre, post` tel que le tableau `pre` contient les numéros préfixes et le tableau `post` contient les numéros postfixes d'un parcours en profondeur du graphe G.
2. En déduire une fonction `acyclique_o : graphe -> bool` qui détermine si un graphe orienté est sans cycle ou non.
3. Écrire une fonction `ordre_topo : graphe -> int array` qui prend en argument un graphe G supposé orienté sans cycle et renvoie un tableau d'entiers correspondant à un ordre topologique du graphe.

   Indication : on pourra soit repartir des tableaux de numéros, soit réécrire un parcours.
4. Écrire une fonction `transpose : graphe -> graphe` qui prend en argument un graphe G orienté et renvoie un nouveau graphe correspondant à G^T.
5. En déduire une fonction `kosaraju : graphe -> int list list` qui prend en argument un graphe orienté et renvoie une liste de listes d'entiers, chaque sous-liste correspondant à une composante fortement connexe du graphe donné en argument.

3 Algorithme de Tarjan

L'algorithme de Tarjan est un algorithme qui, comme l'algorithme de Kosaraju, permet de déterminer les composantes fortement connexes d'un graphe orienté en temps linéaire. L'idée est d'identifier, pour chaque CFC, un sommet représentant la CFC appelé **racine**, correspondant au premier sommet exploré de la CFC (à savoir celui de plus petit numéro préfixe). On introduit pour cela pour chaque sommet s, un numéro $pre_{\min}(s) := \min\{pre(t) \mid t \in CFC(s)\}$. La racine d'un sommet s, $rac(s)$, sera alors le représentant de la CFC de s.

Exercice 3

1. (À faire après le TP) Montrer qu'un sommet $s \in S$ est racine d'une CFC puits de G si et seulement si $pre(s) = pre_{\min}(s)$ et pour chaque autre sommet t accessible depuis s, $pre_{\min}(t) < pre(t)$.

La question précédente donne l'idée d'un algorithme pour trouver et éliminer séquentiellement des CFC puits du graphe. On utilise pour cela une pile qui contiendra les sommets dont on n'a pas encore identifié la CFC. L'algorithme est le suivant :

```
Fonction Tarjan(G)
    k ← 0
    P ← pile vide
    Pour chaque sommet s Faire
        pre(s) ← −1
        rac(s) ← −1
    Pour chaque sommet s Faire
        Si pre(s) = −1 Alors
            TarjanDFS(s)
```

```
Fonction TarjanDFS(s)
    k ← k + 1
    pre(s) ← k
    pre_min(s) ← k
    Empiler s dans P.
    Pour tous les sommets t voisins de s Faire
        Si pre(t) = −1 Alors
            TarjanDFS(t)
            pre_min(s) ← min(pre_min(s), pre_min(t))
        Sinon si rac(t) = −1 Alors
            pre_min(s) ← min(pre_min(s), pre(t))
    Si pre(s) = pre_min(s) Alors
        t ←Dépiler P
        rac(t) ← s
        Tant que s ≠ t Faire
            t ←Dépiler P
            rac(t) ← s
```

2. En utilisant l'algorithme décrit en pseudocode ci-dessus, écrire une fonction de signature `tarjan : graphe -> int list list` qui calcule les CFC d'un graphe orienté donné en argument selon l'algorithme de Tarjan.

Après s'être assuré de la correction des fonctions écrites sur le graphe G_2, on pourra comparer les performances des deux algorithmes.

Exercice 4

On cherche à créer des graphes orientés aléatoires d'ordre n quelconque pour tester les fonctions écrites précédemment. Une manière de faire est de parcourir toutes les arêtes possibles et de conserver chaque arête avec une certaine probabilité. Le problème d'une telle méthode est que la construction d'un graphe se fait en $\Theta(n^2)$, ce qui est trop long pour de grandes valeurs de n.

À la place, on propose de choisir aléatoirement le degré sortant de chaque sommet, puis de choisir aléatoirement les voisins de ce sommet (quitte à faire plusieurs tirages si on tombe sur le même sommet). Si le degré sortant moyen est faible devant n, on gagnera en complexité. Pour choisir le degré aléatoirement, on suggère de simuler une loi normale et d'arrondir à l'entier le plus proche. On admet la propriété suivante :

Proposition

Soient X et Y sont deux variables aléatoires indépendantes de loi uniforme sur $[0, 1]$. On pose :

$$Z = \sqrt{-2\ln X}\cos(2\pi Y)$$

Alors Z suit une loi normale $\mathcal{N}(0, 1)$.

1. Écrire une fonction `normal : float -> float -> float` qui prend en argument deux flottants μ et σ et renvoie un flottant aléatoire suivant une loi normale $\mathcal{N}(\mu, \sigma)$. On rappelle que `Random.float x` renvoie un flottant choisi aléatoirement et uniformément entre 0 et x. On pourra utiliser `Random.self_init ()` pour choisir une graine aléatoire. La constante π n'existe pas par défaut en OCaml, mais on peut la calculer par `4. *. atan 1.`.
2. En déduire une fonction `graphe_alea : int -> float -> float -> graphe` qui prend en argument un entier n et deux flottants μ et σ et renvoie un graphe aléatoire dont les sommets sont

de degré sortant moyen μ avec un écart type σ, suivant une loi normale, arrondi à l'entier le plus proche. On rappelle que `Random.int n` renvoie un flottant choisi aléatoirement et uniformément entre 0 et n.

3. Vérifier que sur un grand nombre de graphes aléatoires, les fonctions `kosaraju` et `tarjan` renvoie des résultats similaires, c'est-à-dire des listes contenant les mêmes composantes fortement connexes (pas nécessairement dans le même ordre).
4. Faire des statistiques comparatives du temps d'exécution des fonctions `kosaraju` et `tarjan` sur des graphes aléatoires de tailles variées. La fonction `Sys.time : unit -> float` renvoie le nombre de secondes écoulées depuis le lancement du fichier et permettra, par une différence, de chronométrer un temps d'exécution. On prendra garde à ne pas compter le temps de création des graphes dans le temps d'exécution pour comparer les algorithmes. On testera avec des graphes d'ordre 50000 et de degré moyen 15 (écart-type au choix).