On obtient les graphes :

```
let g1 = [| [6; 8]; [5]; [3; 8]; [2]; [8];
            [1]; [0; 8]; []; [0; 2; 4; 6] |];;

let g2 = [| [4; 5]; [2; 3; 6]; [7]; [2; 9];
            [3; 5]; []; [8]; [2]; [1; 6]; [0] |];;

let g3 = [| [2; 9]; [8]; [0; 5]; [7]; [9]; [2];
            [8]; [3]; [6]; [0; 4; 10]; [9] |];;

let g4 = [| [7]; [4]; [0; 6]; [1; 2; 5];
            [7]; [4]; [7]; [] |] ;;
```

# 1 Graphes non orientés

**Exercice 1**

1. Cette fonction a déjà été écrite en cours :

```
let composantes_connexes g =
    let n = Array.length g in
    let vus = Array.make n false in
    let lcc = ref [] and cc = ref [] in
    let rec dfs s =
        if not vus.(s) then begin
            vus.(s) <- true;
            cc := s :: !cc;
            List.iter dfs g.(s)
        end in
    for s = 0 to n - 1 do
        if not vus.(s) then
           (dfs s; lcc := !cc :: !lcc; cc := [])
    done;
    !lcc;;
```

2. L'idée est la suivante : on garde en mémoire un tableau de prédécesseurs, initialisé avec des $-1$. Lorsqu'on traite un sommet $s$, pour chacun de ses voisins $t$ :

   si $t$ n'a pas de prédécesseur, on marque $pred[t] = s$ et on relance un parcours depuis $t$ ;

   si $t$ a un prédécesseur et que $pred[s] = t$, on ne fait rien ;

   sinon, c'est qu'il y a un cycle (il existe deux chemins entre $t$ et $s$). On lève une exception pour arrêter le calcul.

```
exception Cyclique;;

let acyclique_no g =
    let n = Array.length g in
    let pred = Array.make n (-1) in
    let rec dfs p s =
        if pred.(s) = -1 then begin
           pred.(s) <- p;
           List.iter (dfs s) g.(s)
        end else if pred.(p) <> s then
           raise Cyclique in
    try for s = 0 to n - 1 do
        if pred.(s) = -1 then dfs s s
    done;
    true
    with Cyclique -> false;;
```

Notons quon aurait pu compter le nombre de composantes connexes $k$ à laide de la fonction précédente, puis compter le nombre darêtes $|A|$, puis vérifier si $|A| \leqslant |S| - k$ ou non.

# 2 Graphes orientés

**Exercice 2**

1. On applique l'algorithme du cours :

```
let numeros_DFS g =
    let n = Array.length g in
    let pre = Array.make n (-1) and
        post = Array.make n (-1) in
    let k = ref 0 in
    let rec dfs s =
        if pre.(s) = -1 then begin
           incr k;
           pre.(s) <- !k;
           List.iter dfs g.(s);
           incr k;
           post.(s) <- !k
        end in
    for s = 0 to n - 1 do
        if pre.(s) = -1 then dfs s
    done;
    pre, post;;
```

2. On teste pour chaque arête $(s, t)$ si $post(s) > post(t)$.

```
let acyclique_o g =
    let n = Array.length g in
    let _, post = numeros_DFS g in
    try for s = 0 to n - 1 do
        let test t =
            if post.(s) < post.(t) then
               raise Cyclique in
        List.iter test g.(s)
    done;
    true
    with Cyclique -> false;;
```

3. On peut calculer le parcours postfixe à partir des numéros postfixes : on crée un tableau de taille $2n+1$, et on indique pour chaque indice quel sommet a cet indice comme numéro postfixe. On peut ensuite parcourir le tableau pour trouver l'ordre postfixe.

```
let ordre_topo g =
    let n = Array.length g in
    let _, post = numeros_DFS g in
    let ordre = Array.make (2 * n + 1) (-1) in
    for s = 0 to n - 1 do
        ordre.(post.(s)) <- s
    done;
    let topo = Array.make n (-1) in
    let s = ref 0 in
    for i = 2 * n downto 1 do
        if ordre.(i) <> -1 then
           (topo.(!s) <- ordre.(i); incr s)
    done;
    topo;;
```

Cela peut être cependant plus simple de relancer un parcours en modifiant un peu la fonction `numeros_DFS` :

```
let ordre_topo g =
    let n = Array.length g in
    let vus = Array.make n false in
    let topo = ref [] in
    let rec dfs s =
        if not vus.(s) then begin
           vus.(s) <- true;
           List.iter dfs g.(s);
           topo := s :: !topo
        end in
    for s = 0 to n - 1 do
        if not vus.(s) then dfs s
    done;
    Array.of_list !topo;;
```

4. On parcourt chaque arête qu'on inverse :

```
let transpose g =
    let n = Array.length g in
    let g' = Array.make n [] in
    for s = 0 to n - 1 do
        let arete t = g'.(t) <- s :: g'.(t) in
        List.iter arete g.(s)
    done;
    g';;
```

5. On applique l'algorithme du cours :

```
let kosaraju g =
    let n = Array.length g in
    let g' = transpose g in
    let topo = ordre_topo g' in
    let vus = Array.make n false in
    let lcc = ref [] and cc = ref [] in
    let rec dfs s =
        if not vus.(s) then begin
            vus.(s) <- true;
            cc := s :: !cc;
            List.iter dfs g.(s)
        end in
    for s = 0 to n - 1 do
        if not vus.(topo.(s)) then
           (dfs topo.(s); lcc := !cc :: !lcc; cc := [])
    done;
    !lcc;;
```

# 3 Algorithme de Tarjan

**Exercice 3**

1. Considérons l'arborescence $T$ d'un DFS de $G$.
   Supposons que $s \in S$ est racine d'une CFC puits de $G$. Par définition de la racine, on en déduit que $pre(s) = pre_{\min}(s)$. Par ailleurs, seuls les sommets de la CFC sont accessible depuis $s$, car c'est une CFC puits. On en déduit que si $t$ est accessible depuis $s$, alors $pre_{\min}(t) = pre(s) < pre(t)$.

   Réciproquement, supposons que $pre(s) = pre_{\min}(s)$ et que chaque sommet $t$ accessible depuis $s$ vérifie $pre_{\min}(t) < pre(t)$. Notons $r$ la racine de la CFC de $s$ et supposons $r \neq s$. Comme $r$ est accessible depuis $s$, on a $pre_{\min}(r) < pre(r)$, ce qui est absurde. On en déduit que $s$ est bien racine de sa CFC. Par ailleurs, comme chaque sommet $t$ accessible depuis $s$ vérifie $pre_{\min}(t) < pre(t)$, on en déduit qu'aucun des sommets accessibles depuis $s$ ne peut être racine de sa CFC, donc la CFC de $s$ est bien une CFC puits.
2. On suit l'algorithme (le résultat est un peu long) :

```
let tarjan g =
    let n = Array.length g in
    let pre = Array.make n (-1) and
        rac = Array.make n (-1) and
        premin = Array.make n (-1) in
    let pile = ref [] and k = ref 0 in
    let lcc = ref [] and cc = ref [] in
    let rec tarjan_dfs s =
        incr k;
        pre.(s) <- !k;
        premin.(s) <- !k;
        pile := s :: !pile;
        let traiter t =
            if pre.(t) = -1 then begin
               tarjan_dfs t;
               premin.(s) <- min premin.(s) premin.(t)
            end else if rac.(t) = -1 then
               premin.(s) <- min premin.(s) pre.(t) in
        List.iter traiter g.(s);
        if pre.(s) = premin.(s) then begin
           let t = ref (List.hd !pile) in
           pile := List.tl !pile;
           rac.(!t) <- s;
           cc := !t :: !cc;
           while s <> !t do
              t := List.hd !pile;
              pile := List.tl !pile;
              rac.(!t) <- s;
              cc := !t :: !cc;
           done;
           lcc := !cc :: !lcc;
           cc := [] end in
    for s = 0 to n - 1 do
        if pre.(s) = -1 then tarjan_dfs s
    done;
    !lcc;;
```

**Exercice 4**

1. On suit les formules.

```
Random.self_init ();;

let pi = 4. *. atan 1.;;
let normal mu sigma =
    let x = Random.float 1. and y = Random.float 1. in
    let z = sqrt (-2. *. log x) *. cos (2. *. pi *. y) in
    sigma *. z +. mu;;
```

2. On choisit le degré de chaque sommet aléatoirement, puis on choisit ses voisins aléatoirement. On fait des tirages aléatoires tant qu'on n'a pas le bon degré (on suppose que le degré est très inférieur au nombre de sommets).

```
let graphe_alea n mu sigma =
    let g = Array.make n [] in
    for s = 0 to n - 1 do
        let x = normal mu sigma in
        let deg = int_of_float (floor (x +. 0.5)) in
        let len = ref 0 in
        while !len < deg do
            let t = Random.int n in
            if s <> t && not (List.mem t g.(s)) then
               (g.(s) <- t :: g.(s); incr len)
        done
    done;
    g;;
```

3. Pour pouvoir vérifier que les résultats sont similaires, ils faut trier les listes de listes obtenues pour tester leur égalité. On utilise pour cela `List.sort` (la fonction `compare` est la fonction de comparaison par défaut).

```
let trier llst =
    List.sort compare (List.map (List.sort compare) llst);;
```

On peut alors comparer avec un grand nombre de graphes :

```
for n = 30 to 1000 do
    let g = graphe_alea n 5. 1. in
    assert (trier (kosaraju g) = trier (tarjan g))
done;;
```

4. On écrit une fonction de chronomètre :

```
let temps f x =
    let debut = Sys.time () in
    let fin = ignore (f x); Sys.time () in
    fin -. debut;;
```

On peut alors comparer les temps d'exécution :

```
let tps_kos = ref 0. and
    tps_tar = ref 0. in
for i = 0 to 10 do
    let g = graphe_alea 50000 15. 5. in
    tps_kos := !tps_kos +. temps kosaraju g;
    tps_tar := !tps_tar +. temps tarjan g
done;
!tps_kos, !tps_tar;;
```

On constate un facteur 2 en faveur de l'algorithme de Tarjan.