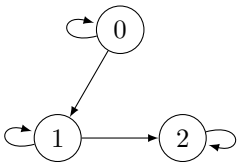
I Représentations

1. Quel est le nombre de chemins de longueur 100 de 0 à 2 dans le graphe orienté suivant?



Solution : Sa matrice d'adjacence est

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

Alors $A^n = (\underbrace{\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}}_{I} + \underbrace{\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}}_{J})^n = I + nJ + \binom{n}{2} J^2 = \begin{pmatrix} 1 & n & \binom{n}{2} \\ 0 & 1 & n \\ 0 & 0 & 1 \end{pmatrix}$.

Le nombre demandé est donc $\boxed{\binom{100}{2}}$.

2. Soit $\vec{G} = (V, \vec{E})$ un graphe orienté représenté par une matrice d'adjacence `m`.
Écrire une fonction `trou_noir m` renvoyant en $O(|V|)$ un sommet t vérifiant :
   - $\forall u \neq t : (u, t) \in \vec{E}$
   - $\forall v \neq t : (t, v) \notin \vec{E}$

   Si $\vec{G}$ n'a pas de trou noir, on pourra renvoyer `-1`.

Solution : On conserve un candidat c pour être trou noir (initialement 0) que l'on compare au sommet suivant v : s'il y a un arc (c, v) alors v devient le candidat.

```
let rec trou_noir g =
    let n = Array.length g in
    let c = ref 0 in (* candidat trou noir *)
    for v = 1 to n - 1 do
        if g.(!c).(v) = 1 then c := v
    done;
    (* verification que c est un trou noir *)
    for v = 0 to n - 1 do
        if !c <> -1 && v <> !c &&
           (g.(!c).(v) = 1 || g.(v).(!c) = 0)
        then c := -1
    done;
    !c
```

Si `g` possède un trou noir `t`, alors, quand `v = t` dans le premier `for`, `c` prend la valeur `t` (car `g.(!c).(t) = 1`, puisque `t` est un trou noir).
Par la suite, `c` ne peut plus changer de valeur (sinon cela contredirait le fait que `t` est un trou noir).
Donc :

- S'il existe un trou noir, celui-ci va bien être renvoyé par `trou_noir g`.
- S'il n'y a pas de trou noir, on va passer dans le `if` du deuxième `for` et renvoyer `-1`.

`trou_noir g 0 1` renvoie le seul sommet qui peut être un trou noir.

3. Écrire une fonction

```
pere_to_arb : int array -> int arb
```

qui transforme en temps linéaire un arbre représenté par un tableau `pere` (où `pere.(i)` est le prédécesseur du sommet `i`)

en un arbre persistant de type :

```
type 'a arb = N of 'a * 'a arb list
```

Si `r` est la racine, on supposera que `pere.(r) = r`.
Remarque : l'arbre peut avoir un nombre quelconque de fils, d'où l'utilisation d'une liste pour les fils.

Solution : On peut construire un tableau des fils : `fils.(i)` va être la liste des fils du sommet `i`. Il est ensuite facile d'en déduire l'arbre correspondant. On utilise `map f [u0; u1; ...]` qui renvoie `[f u0; f u1; ...]` :

```
let arb_of_pere pere =
    let n = Array.length pere includegraphics
    let r = ref 0 in (* contiendra la racine *)
    for i = 0 to n - 1 do
        if pere.(i) = i then r := i
        else fils.(pere.(i)) <- i::fils.(pere.(i))
    done;
    let rec fils_to_arb i = (* renvoie l'arbre enraciné en i *)
        N(i, List.map fils_to_arb fils.(i)) in
    fils_to_arb !r
```

II Distances

Soit $G = (V, E)$ un graphe représenté par liste d'adjacence. On rappelle que la **distance** de u à v est la longueur minimum d'un chemin de u à v (c'est aussi une distance au sens mathématiques, pour un graphe non-orienté).

1. L'**excentricité** d'un sommet u est la distance maximum de ce sommet à un autre. Écrire une fonction `exc : int list array -> int -> int` renvoyant en $O(|V| + |E|)$ l'excentricité d'un sommet.
2. Solution : On peut renvoyer la distance du dernier sommet visité par un BFS, ou tout simplement la distance max dans le tableau calculé depuis un BFS partant de `u` :

```
let exc g u =
    let n = Array.length g in
    let dist = Array.make n (-1) in
    let q = Queue.create () in
    let add d v =
        if dist.(v) = -1 then (
            dist.(v) <- d;
            Queue.add (v, d) q
        ) in
    add 0 r;
    while not (Queue.is_empty q) do
        let u, d = Queue.pop q in
        List.iter (add (d + 1)) g.(u)
    done;
    let m = ref 0 in (* sommet de distance max *)
    for i = 0 to n - 1 do
        if dist.(i) > dist.(!m)
        then m := i
    done;
    !m
```

3. Écrire une fonction `diametre : int list array -> int` renvoyant en $O(|V| \times (|V| + |E|))$ le **diamètre** d'un graphe, c'est à dire la distance maximum entre deux sommets.
4. Solution : On cherche l'excentricité maximum :

```
let diametre g =
    let n = Array.length g in
    let m = ref 0 in (* excentricité max *)
    for v = 0 to n - 1 do
        m := max !m (exc g v) in
    done;
    !m
```

5. Écrire une fonction `centre : int list array -> int` renvoyant en $O(|V| \times (|V| + |E|))$ le **centre** d'un graphe, c'est à dire le sommet d'excentricité minimum.

   Solution :

```
let centre g =
    let n = Array.length g in
    let u = ref 0 in
    let u_exc = ref (exc g 0) in
    for v = 0 to n - 1 do
        let v_exc = exc g v in
        if v_exc < !u_exc then (
            u := v;
            u_exc := v_exc
        )
    done;
    !u
```

6. Peut-on améliorer les trois algorithmes précédents si G est un arbre?

   Solution : On peut trouver le diamètre d'un arbre en $O(|V| + |E|)$:

   - 1ère méthode : convertir G en `int arb` (avec `arb_of_pere` du I appliqué sur le tableau des pères d'un parcours de G) puis utiliser le fait que le diamètre de `N(r, a ::q)` est soit le diamètre de `a`, soit le diamètre de `N(r, q)`, soit la hauteur de `a` + la hauteur de `N(r, q)`. Puisqu'on a besoin du diamètre et de la hauteur, on renvoie un couple (diamètre, hauteur).

```
(* renvoie (diamètre de a, hauteur de a) *)
let rec diametre a = match a with
    | N(r, []) -> 0, 0
    | N(r, e::q) -> let de, he = diametre e in
                    let dq, hq = diametre (N(r, q)) in
                    max de (max dq (he + hq + 1)), max (he + 1) hq
```

   - 2ème méthode : faire un BFS depuis un sommet quelconque (ici 0), puis refaire un BFS depuis le dernier sommet visité. On peut montrer que la dernière distance obtenue est le diamètre (ceci ne marche qu'avec un arbre).

```
let bfs1 g =
  let vu = make_vect g.n false in
  let rec aux v cur next = match cur with
    | [] -> if next = [] then v else aux v next []
    | e::q when vu.(e) -> aux e q next
    | e::q -> (vu.(e) <- true;
               aux e q next) in
  aux 0 [0] [];;
let diam g = exc g (bfs1 g);;
```

   On peut montrer que le milieu d'un chemin réalisant le diamètre d'un arbre est un centre (exercice). On pourrait donc calculer le centre d'un arbre en temps linéaire.

7. Soient $S \subset V$ et $T \subset V$. Comment calculer efficacement la distance entre S et T, c'est à dire la distance minimum entre un sommet de S et un de T?

   ▶ 1ère solution : rajouter deux sommets s et t reliés à tous les sommets de S et T respectivement. La distance entre S et T est alors celle entre s et t moins 2.

   2ème solution : faire un BFS en initialisant la couche `cur` en cours avec tous les sommets de S. On s'arête dès qu'on trouve un sommet de T.

8. Soient $u, v, w \in V$. Comment trouver efficacement un plus court chemin de u à w passant par v?
   ▶ On peut faire un BFS depuis v pour en déduire un plus court chemin de u à v et un plus court chemin de v à w, qu'on concatène.

9. Soit $G = (V, E)$ et $k \in \mathbb{N}$ tel que $\deg(v) \leq k$, $\forall v \in V$. Soient $u, v \in V$. Expliquer comment trouver la distance d de u à v en $O(\sqrt{k^d})$. Comment procéder pour un graphe orienté?

   ▶ Un BFS va visiter au plus $1 + k + k^2 + ... + k^p = \dfrac{k^{p+1} - 1}{k - 1} = O(k^p)$ sommets à profondeur p. On peut faire partir simultanément deux BFS : un depuis u et l'autre depuis v. Au moment où ils se « rejoignent », la somme des profondeurs des BFS est égale à la distance de u à v. Le nombre total de sommets visités est au plus $O(k^{\frac{d}{2}}) + O(k^{\frac{d}{2}}) = O(k^{\frac{d}{2}})$.

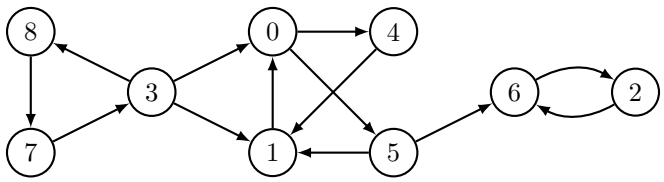
III Composantes fortement connexes

Dans tout l'exercice, `g : int list array` est un graphe orienté représenté par liste d'adjacence.

III.1 Tri topologique

1. Écrire une fonction `post_dfs g vu r` renvoyant la liste des sommets atteignables depuis `r` dans l'ordre de fin de traitement croissant d'un DFS (c'est à dire dans l'ordre postfixe/suffixe de l'arbre de parcours en profondeur). `vu` est un tableau des sommets déjà visités. On pourra utiliser `@` pour simplifier l'écriture.

```
let rec post_dfs g vu r =
  if vu.(r) then []
  else (vu.(r) <- true;
        let rec do_voisins = function
          | [] -> [r]
          | v::q -> (post_dfs g vu v) @ (do_voisins q) in
        do_voisins g.(r));;
```



2. Quelle est la liste renvoyée par `post_dfs g vu 0` si `g` est le graphe ci-dessus?
   ▶ En traitant les sommets par numéro croissant, lorsqu'il y a plusieurs possibilités : `[1; 4; 2; 6; 5; 0]`.

3. Soit `[v0; v1; ...]` la liste renvoyée par `post_dfs g vu r`. On suppose `g` sans cycle.
   Montrer que : (`vi`, `vj`) est un arc de `g` $\Longrightarrow$ `i > j`.
   ▶ On distingue deux possibilités :

   - Si `post_dfs g vu vj` est exécuté avant `post_dfs g vu vi` : il ne peut pas y avoir de chemin de `vj` vers `vi` (sinon il y aurait un cycle) donc `post_dfs g vu vj` doit être fini avant que `post_dfs g vu vi` ne commence. Donc `i > j`.
   - Sinon : puisque (`vi`, `vj`) est un arc, `post_dfs g vu vi` va visiter `vj`. `post_dfs g vu vi` va donc se terminer après `post_dfs g vu vj`. Donc `i > j`.

4. On suppose `g` sans cycle. En déduire une fonction `tri_topo g` effectuant un **tri topologique** de `g`, c'est à dire renvoyant une liste `[v0; v1; ...]` de tous ses sommets de façon à ce que : (`vi`, `vj`) est un arc de `g` $\Longrightarrow$ `i < j`.

   ▶ Il suffit d'appliquer plusieurs fois `post_dfs` puis d'inverser la liste obtenue :

```
let tri_topo g =
  let n = vect_length g in
  let vu = make_vect n false in
  let rec aux v =
    if v = n then [r]
    else post_dfs g vu v @ aux (v+1) in
  rev (aux 0);;
```

   Remarque : on peut voir le tri topologique comme une généralisation d'un tri classique, où $a \leq b$ est remplacée par $a \to b$. On pourrait trier des entiers en appelant `tri_topo` sur le graphe correspondant, mais le nombre d'arcs serait quadratique, donc la complexité aussi.

Application : on veut savoir dans quel ordre effectuer des tâches (les sommets) dont certaines doivent être effectuées après d'autres (arcs = dépendances). Par exemple pour résoudre un problème par programmation dynamique, on peut construire le graphe dont les sommets sont les sous-problèmes, un arc (u, v) indiquant que la résolution de v nécessite celle de u. Il faut alors résoudre les sous-problèmes dans un ordre topologique.

III.2 Algorithme de Kosaraju

1. Écrire une fonction

   `tr : int list array -> int list array`

   renvoyant la **transposée** d'un graphe, obtenue en inversant le sens de tous les arcs.

```
let tr g =
  let n = vect_length g in
  let res = make_vect n [] in
  for u = 0 to n - 1 do
    do_list (fun v -> res.(v) <- u::res.(v)) g.(u)
  done;
  res;;
```

L'algorithme de Kosaraju consiste à trouver les composantes fortement connexes de `g` de la façon suivante :

(i) appliquer plusieurs DFS sur `g` jusqu'à avoir visité tous les sommets, en calculant la liste `l` des sommets de `g` dans l'ordre de fin de traitement décroissant.

(ii) faire un DFS dans `tr g` depuis le premier sommet `r` de `l` : l'ensemble des sommets atteints est alors une composante fortement connexe de `g`.

(iii) répéter (ii) tant que possible en remplaçant `r` par le prochain sommet non visité de `l`.

2. Appliquer la méthode sur le graphe de la question III.1.1.

3. Écrire une fonction `kosaraju g` renvoyant la liste des composantes fortement connexes de `g` (chaque composante fortement connexe étant une liste de sommets).
   ▶ La liste de (i) est exactement celle renvoyée par `tri_topo`.

```
let kosaraju g =
  let n = vect_length g in
  let vu, tg = make_vect n false, tr g in
  let rec dfs2 = function
    | [] -> []
    | v::q -> (post_dfs tg vu v)::dfs2 q in
  dfs2 (tri_topo g);;
```

4. Quelle serait la complexité en évitant l'utilisation de `@`?
   ▶ Le calcul de transposée est en $O(|V| + |E|)$, de même que les 2 DFS « complets ». La complexité totale serait donc 3 $\times$ $O(|V| + |E|) = O(|V| + |E|)$.