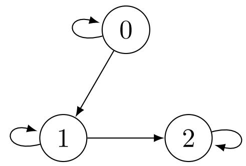
I Représentations

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



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`.
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.

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).

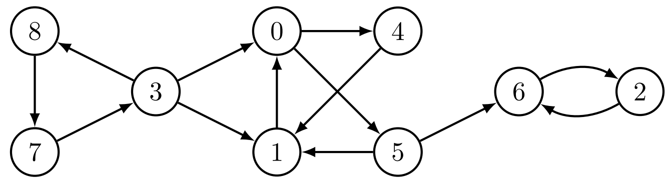
1. L'**excentricité** d'un sommet u est la distance maximum de ce sommet à un autre. Écrire une fonction `exc : int list array -> int -> int` renvoyant en $O(|V| + |E|)$ l'excentricité d'un sommet.
2. Écrire une fonction `diametre : int list array -> int` renvoyant en $O(|V| \times (|V| + |E|))$ le **diamètre** d'un graphe, c'est à dire la distance maximum entre deux sommets.
3. 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.
4. Peut-on améliorer les trois algorithmes précédents si G est un arbre?
5. 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?
6. Soient $u, v, w \in V$. Comment trouver efficacement un plus court chemin de u à w passant par v?
7. 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?

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.
2. Quelle est la liste renvoyée par `post_dfs g vu 0` si `g` est le graphe ci-dessus?



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`.

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`.
   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.

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).

4. Quelle serait la complexité en évitant l'utilisation de `@`?