I Représentations

1. Écrire des fonctions `mat_of_list : int list array -> int array array` et `list_of_mat : int array array -> int list array` pour passer de liste d'adjacence à matrice d'adjacence et inversement. Quelles sont leurs complexités?

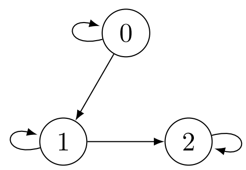
2. Soit $\overrightarrow{G} = (V, \overrightarrow{E})$ un graphe orienté représenté par une matrice d'adjacence `m : int array array`.
   Écrire une fonction `trou_noir m` renvoyant en $O(|V|)$ un sommet t vérifiant:

   - $(u, t) \in \overrightarrow{E}, \forall u \neq t$
   - $(t, v) \notin \overrightarrow{E}, \forall v \neq t$

   On supposera dans un premier temps que ce sommet existe, puis on expliquera comment le déterminer.

3. Écrire une fonction `arb_of_pere : int array -> int arb` qui transforme en temps linéaire un arbre représenté par un tableau des pères (par exemple l'arbre de parcours en largeur/profondeur) en un arbre persistant (`type 'a arb = N of 'a * 'a arb list`). La racine est son propre père.

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



5. Comment déterminer si un graphe possède un cycle de longueur k en utilisant sa matrice d'adjacence A? On en déduit par exemple un algorithme pour déterminer si un graphe est sans triangle (cf TD 1).

6. Comment déterminer le nombre de chemins **élémentaires** (qui ne reviennent pas sur le même sommet) de longueur k en utilisant la matrice d'adjacence?

7. Soit G un graphe non-orienté ***k*-régulier** (dont tous les sommets ont degré k). Montrer que k est valeur propre de la matrice d'adjacence A de G. Quelle propriété similaire a-t-on pour les graphes orientés?

8. Quelles sont les valeurs propres possibles de la matrice d'adjacence A d'un graphe orienté sans cycle?

II Distances

Soit $G = (V, E)$ un graphe. 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 graph -> int -> int` renvoyant en $O(|V| + |E|)$ l'excentricité d'un sommet.

2. Écrire une fonction `diametre : int graph -> 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 graph -> 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?

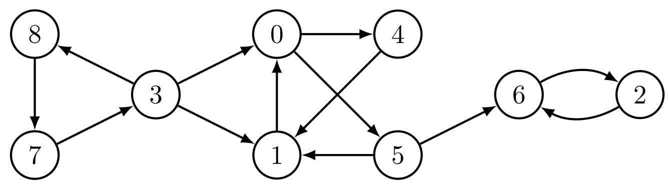
7. Soit $G = (V, E)$ et $k \in \mathbb{N}$ tel que $\deg(v) \leq k$, $\forall v \in V$. Soient $u, v \in V$. Expliquer comment trouver la distance d de u à v en $O(\sqrt{k^d})$. Comment procéder pour un graphe orienté?

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 au dessus.
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 `@`?

Nous verrons dans le cours de logique une très jolie application à la résolution du problème 2-SAT.

IV Graphe biparti

Un graphe $G = (V, E)$ est **biparti** si $V = A \sqcup B$ et toute arête a une extrémité dans A, une dans B (on peut colorier ses sommets de deux couleurs tel que toute arête ait ses extrémités de couleurs différentes).

1. Écrire une fonction `biparti g` renvoyant un tableau de couleurs (0 ou 1) des sommets si `g` est biparti, qui déclenche une exception sinon. On supposera `g` connexe.

2. Montrer qu'un graphe est biparti ssi il ne contient pas de cycle de longueur impaire (on en déduit par exemple qu'un arbre est biparti...).

3. Caractériser les matrices d'adjacences des graphes bipartis.

4. Soit M matrice d'adjacence d'un graphe biparti. Montrer que $\lambda \in Sp(M) \iff -\lambda \in Sp(M)$.

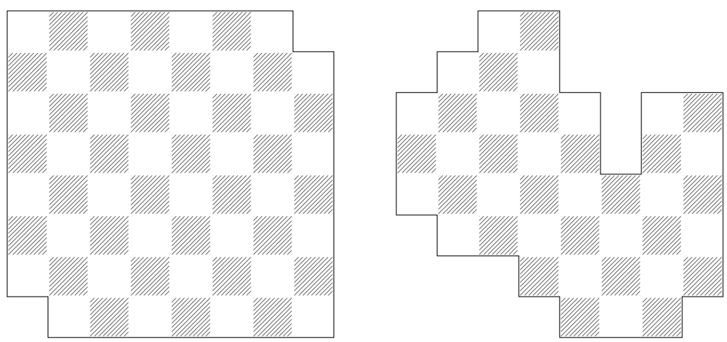
Un **couplage** dans un graphe est un ensemble d'arêtes dont toutes les extrémités sont différentes (si les sommets sont des personnes et les arêtes des affinités, ceci revient à former des couples). Un couplage est **parfait** s'il contient tous les sommets.

5. (Théorème des mariages (Hall)) Soit $G = (A \sqcup B, E)$ un graphe biparti. Si $X \subseteq A$, on note $N(X)$ l'ensemble des voisins des sommets de X. Montrer que:

$$G \text{ a un couplage parfait}^{1} \iff |N(X)| \geq |X|,\ \forall X \subseteq A$$

   Indice: pour $\Longleftarrow$, supposer dans un premier temps $|N(X)| > |X|$, $\forall X \subseteq A$.

6. Application: on considère un échiquier auquel on a enlevé des cases et on veut savoir s'il est possible de le paver (c'est à dire recouvrir sans chevauchement) avec des dominos de taille 2×1. Est-ce possible avec les suivants?



7. Montrer que tout graphe ***k*-régulier** (dont tous les sommets ont degré k) possède un couplage parfait.

Soit $G = (A \cup B, E)$ un graphe **biparti complet** ($\{u, v\} \in E \iff u \in A \ \&\& \ v \in B$) tel que $|A| = |B|$ et dont les sommets sont des points de $\mathbb{R}^2$ en position générale (3 sommets quelconques ne peuvent pas être alignés).

8. Montrer que G possède un couplage parfait **planaire**, c'est à dire sans aucun croisement d'arête.
   Indice: (1ère solution) partir d'un couplage parfait quelconque puis « décroiser ».
   (2ème solution) commencer par prouver qu'il existe une droite séparant le problème en deux sous-problèmes [2].

9. En déduire un algorithme « efficace » pour trouver un tel couplage. Quelle est sa complexité?

[1] Nous verrons plus tard un algorithme efficace pour trouver ce couplage

[2] C'est un cas particulier du très sérieux théorème du sandwich au jambon