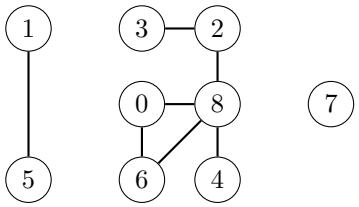
Dans tout le TP, on travaille avec des graphes représentés par tableaux de listes d'adjacence.

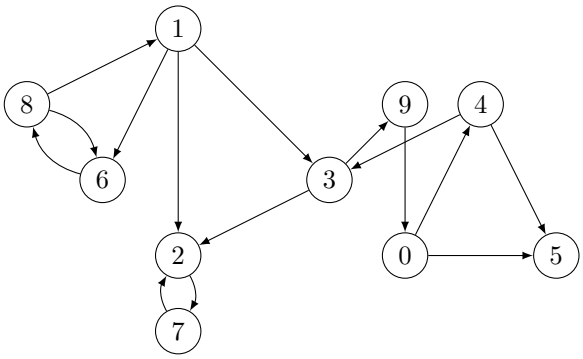
```
type graphe = int list array
```

Exercice 0

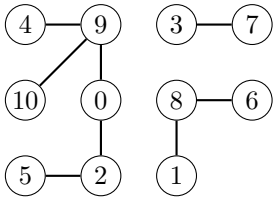
Créer des variables `g1`, `g2`, `g3` et `g4` correspondant aux tableaux de listes d'adjacence des graphes suivants :



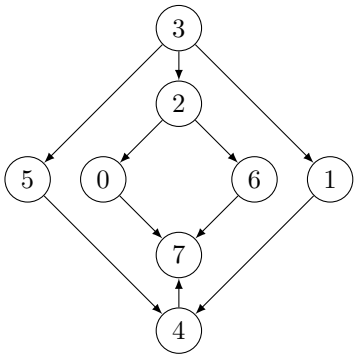
Graphe G_1



Graphe G_2



Graphe G_3



Graphe G_4

On utilisera ces graphes pour tester les différentes fonctions.

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 (g: 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 (g: 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 réutiliser la fonction précédente.

2 Graphes orientés

Exercice 2

1. Écrire une fonction `numeros_DFS (g: 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 (g: graphe) : bool` qui détermine si un graphe orienté est sans cycle ou non.

Palier 1

3. Écrire une fonction `ordre_topo (g: graphe) : int array` qui prend en argument un graphe orienté G et renvoie un tableau d'entiers correspondant à un ordre inverse d'un parcours en profondeur postfixe du graphe. En particulier, si le graphe est sans cycle, l'ordre renvoyé doit correspondre à un ordre topologique.

   Indication : on pourra soit repartir des tableaux de numéros, soit réécrire un parcours.
4. Écrire une fonction `transpose (g: 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 (g: 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. Les composantes fortement connexes devront être rangées dans la liste selon un ordre topologique du métagraphe.

3 Résolution de 2-SAT

On cherche à résoudre efficacement le problème 2-SAT. On propose d'étudier dans un premier temps une résolution naïve, puis une résolution efficace en utilisant l'algorithme de Kosaraju.

On suppose disposer d'un ensemble de variables $\mathcal{V} = \{x_0, \dots, x_{n-1}\}$. On représente une formule en 2-FNC en utilisant les types suivants :

```
type litteral = Neg of int | Pos of int
type clause = litteral * litteral
type deux_fnc = clause list
```

Ainsi, `Neg i` représente le littéral $\neg x_i$ (ou $\overline{x_i}$) et `Pos i` représente x_i. Une clause disjonctive de taille 2 est donnée par le couple de ses littéraux, et une formule en 2-FNC par la liste de ses clauses disjonctives.

Par exemple, la formule $\varphi_0 = (\overline{x_1} \vee \overline{x_2}) \wedge (\overline{x_0} \vee x_2) \wedge x_2 \wedge (x_1 \vee x_0) \wedge (\overline{x_1} \vee x_2)$ peut être définie par :

```
let phi0 = [(Neg 1, Neg 2); (Neg 0, Pos 2); (Pos 2, Pos 2); (Pos 1, Pos 0); (Neg 1, Pos 2)]
```

On note que la clause (x_2) a été réécrite en $(x_2 \vee x_2)$.

0. Créer la formule φ_0 ainsi que la formule $\varphi_1 = (\overline{x_0} \vee x_3) \wedge (x_0 \vee \overline{x_2}) \wedge (x_1 \vee \overline{x_2}) \wedge (x_0 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_0}) \wedge (\overline{x_3} \vee x_2)$.

On représente une valuation (ou distribution de vérité) μ par un tableau de taille n contenant des 0 et des 1.

```
type valuation = int array
```

Ainsi, si μ est représenté par un tableau `mu`, alors $\mu(x_i)$ est égal à `mu.(i)`.

1. Écrire une fonction `eval_litt (mu: valuation) (litt: litteral) : int` qui prend en argument une valuation et un littéral et évalue ce littéral.
2. En déduire une fonction `evalue (mu: valuation) (phi: deux_cnf) : int` qui prend en argument une valuation μ et une formule φ en 2-FNC et calcule $\mu(\varphi)$.

On résoudre naïvement le problème 2-SAT, on veut parcourir l'ensemble des 2^n valuations possibles.

3. Écrire une fonction `incremente (mu: valuation) : bool` qui prend en argument une valuation et modifie le tableau pour qu'il devienne la valuation suivante selon l'ordre lexicographique. La fonction devra renvoyer `true` si la valuation n'était pas la plus grande selon l'ordre lexicographique et `false` sinon.

   Par exemple, `incremente [|1;0;0;1;1|]` modifie le tableau en `[|1;0;1;0;0|]` et renvoie `true`, et `incremente [|1;1;1;1|]` modifie le tableau en `[|0;0;0;0|]` et renvoie `false`.
4. En déduire une fonction `modele (n: int) (phi: deux_cnf) : valuation option` qui prend en argument un entier n et une formule φ en 2-FNC sur n variables et renvoie `None` si φ n'est pas satisfiable, et `Some mu` avec μ un modèle de φ sinon.

Palier 2

5. Écrire une fonction `graphe_implication (n: int) (phi: deux_cnf)` qui prend en argument un entier n et une formule φ en 2-FNC sur n variables et renvoie son graphe d'implication. On encodera un littéral x_i par le sommet i et un littéral $\overline{x_i}$ par le sommet $i+n$.

Pour tester la satisfiabilité d'une formule en 2-FNC à partir de son graphe d'implication, il peut être plus simple de manipuler un tableau qui indique le numéro de composante fortement connexe pour chaque sommet plutôt qu'une liste des composantes fortement connexes.

6. Écrire une fonction `convertir (k: int) (lcfc: int list list) : int array` qui prend en argument un entier k correspondant au nombre de sommets d'un graphe orienté $G = (S, A)$ et une liste des composantes fortement connexes de G et renvoie un tableau d'entiers `cfc` de taille k tel que pour tout sommet $s \in S$, `cfc.(s)` est égal au numéro de la composante fortement connexe de s, selon une numérotation avec des entiers consécutifs commençant à zéro.
7. En déduire une fonction `deux_sat (n: int) (phi: deux_cnf) : bool` qui résout efficacement le problème 2-SAT.

Lorsque la formule est effectivement satisfiable, il est préférable d'avoir un modèle de la formule.

8. Écrire une fonction `modele2 (n: int) (phi: deux_cnf) : valuation option` qui prend en argument un entier n et une formule φ en 2-FNC sur n variables et renvoie `None` si φ n'est pas satisfiable, et `Some mu` avec μ un modèle de φ sinon.

4 Lecture de fichiers et vérifications

On dispose de deux fichiers contenant des formules en 2-FNC : un fichier de formules satisfiables et un fichier de formules non satisfiables. On souhaite les utiliser pour vérifier la correction des fonctions précédentes.

Chaque fichier contient une ligne par formule, une formule ayant le format suivant :

- elle commence par un entier n correspondant au nombre de variables de la formule, suivi d'une espace ;
- ensuite, on trouve une description de la formule elle-même, correspondant à des clauses séparées par des points-virgules, chaque clause étant deux littéraux séparés par une virgule, chaque littéral commençant par `+` (littéral positif) ou `-` (littéral négatif), suivi du numéro de variable.

Par exemple, la formule $\varphi_0 = (\overline{x_1} \vee \overline{x_2}) \wedge (\overline{x_0} \vee x_2) \wedge x_2 \wedge (x_1 \vee x_0) \wedge (\overline{x_1} \vee x_2)$ correspondrait à la ligne :

```
3 -1,-2;-0,+2;+2,+2;+1,+0;-1,+2
```

On utilisera les fonctions suivantes pour faire de la lecture de fichier :

- `open_in : string -> in_channel` prend en argument un nom de fichier et ouvre un canal de lecture correspondant au fichier ;
- `close_in : in_channel -> unit` ferme un canal de lecture ;
- `input_line : in_channel -> string` prend en argument un canal de lecture et renvoie, sous forme de chaîne de caractères, la prochaine ligne du fichier. La fonction lève une exception `End_of_file` si on est arrivé à la fin du fichier ;
- `String.split_on_char : char -> string -> string list` prend en argument un caractère `c` et une chaîne de caractères `s` et renvoie une liste de chaînes correspondant aux facteurs de `s` séparés par le caractère `c`. Par exemple, `String.split_on_char ',' "ab,cdef,gh"` renvoie `["ab"; "cdef"; "gh"]`.

1. Écrire une fonction `lire_formule (str: string) : deux_fnc` qui prend en argument une chaîne de caractères correspondant à une formule (sans le nombre de variables) et renvoie la formule associée. Par exemple, `lire_formule "-1,-2;-0,+2;+2,+2;+1,+0;-1,+2"` renvoie :

```
[(Neg 1, Neg 2); (Neg 0, Pos 2); (Pos 2, Pos 2); (Pos 1, Pos 0); (Neg 1, Pos 2)]
```

2. En déduire une fonction `lire_fichier (fichier: string) : (int * deux_fnc) list` qui prend en argument un nom de fichier et renvoie une liste de couples de la forme (n, φ) où chaque ligne du fichier correspond à une formule φ à n variables.
3. Vérifier la correction des fonctions `deux_sat` et `modele2`.

Palier 3