Ce TP est fortement inspiré du sujet des Mines 2003.
Les notations mathématiques sont notées dans une police *en italique*, et les instances correspondantes en Caml sont notées dans une police `sans sérif`. Par exemple, si x est une quantité mathématique, son codage sous Caml sera noté `x`. La police **en gras** sert dans les définitions à noter les termes qui sont définis.

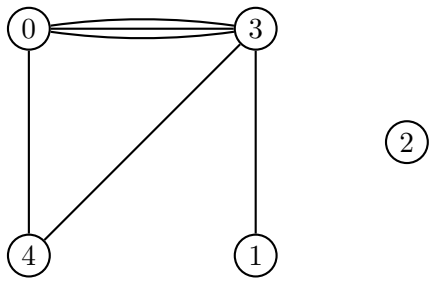
Codage d'un graphe

Quelques définitions et notations :

- Un **graphe** g est un couple $(v(g), e(g))$ où $v(g)$ est un ensemble fini dont les éléments sont appelés les **sommets** de g, et $e(g)$ est une liste de paires d'éléments de $v(g)$, chaque paire s'appelant une **arête** du graphe g.
- Le cardinal de $v(g)$ se note $n(g)$, et nous numéroterons dans toute la suite les sommets de g par les entiers entre 0 et $n(g)-1$: ainsi $v(g) = [|0, n(g)-1|]$, et le graphe g est entièrement déterminé par le couple $(n(g), e(g))$.
- Deux sommets s et t de g sont dits **voisins** si $\{s, t\}$ est une arête de g.
- Un sommet est dit **isolé** s'il ne possède aucun voisin.
- Pour coder le graphe g nous utiliserons les types à champs suivants :

```
type arete = {a : int; b : int};;
type graphe = {n : int; aretes : arete list};;
```

Exemple : le graphe $gex1$ suivant et le code qui le définit :

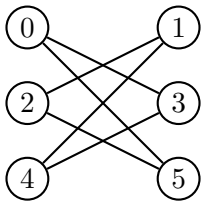



```
let gex1 = {n = 5;
            aretes = [{a = 0; b = 4};
                      {a = 4; b = 3};
                      {a = 0; b = 3};
                      {a = 1; b = 3};
                      {a = 0; b = 3};
                      {a = 3; b = 0}]};;
```

Remarquer sur l'exemple précédent que deux sommets peuvent être liés par plusieurs arêtes. Remarquer aussi que le sommet 2 est isolé.

1) Rentrer le code qui définit le graphe $gex1$ (il servira à des tests).

2) Coder le graphe $gex2$ suivant :



3) Écrire une fonction :

```
insere : int list -> int -> int list
```

spécifiée ainsi : si `li` est une liste d'entiers triés par ordre croissant et si `s` est un entier, alors `insere li s` renvoie :

— la liste d'entiers obtenue en insérant `s` selon l'ordre croissant si la valeur `s` ne figure pas dans `li` ;
— la liste `li` si `s` figure déjà dans `li`.

La complexité de cette fonction doit être linéaire par rapport à la taille de la liste `li`.

4) Écrire une fonction :

```
voisins : graphe -> int -> int list
```

qui calcule la liste triée par ordre croissant des voisins d'un sommet dans un graphe. On utilisera la fonction `insere`.

Un algorithme simple de bonne coloration

On appelle **coloration** d'un graphe g une application c définie sur l'ensemble $v(g)$ des sommets de g, et à valeurs dans $\mathbb{N}^*$. Pour tout sommet s de g, l'entier $c(s)$ s'appelle la couleur de s pour la coloration c.

Une coloration c d'un graphe g est une **bonne coloration** de g si pour toutes arêtes $\{s,t\} \in e(g)$, on a $c(s) \neq c(t)$; autrement dit, une coloration est une bonne coloration si les extrémités de toute arête sont de couleurs différentes.

Une coloration sera codée par un tableau d'entiers : les indices du tableau correspondent aux sommets et les valeurs codées dans le tableau correspondent aux couleurs des sommets.

5) Écrire une fonction :

```
bonne_coloration : graphe -> int array -> bool
```

qui permet de déterminer si une coloration est bonne ou non. On supposera que la taille du tableau passé en paramètre correspond au nombre de sommets du graphe passé en paramètre, on ne demande pas de la vérifier.

On considère l'algorithme de bonne coloration suivante : les couleurs disponibles sont les entiers naturels non nuls ; la plus petite couleur est 1. On colorie les sommets les uns après les autres, dans l'ordre $0, 1, 2 \ldots n(g) - 1$; lorsqu'on considère un sommet s, on lui attribue la plus petite couleur disponible, c'est–à–dire la plus petite couleur qui n'est pas déjà la couleur d'un voisin de s.

6) Appliquer à la main l'algorithme de bonne coloration ci–dessus pour le graphe $gex2$. Montrer qu'il nécessite 3 couleurs. Montrer qu'il existe une autre bonne coloration de $gex2$ qui ne nécessite que 2 couleurs.

7) Construire à la main un exemple de graphe pour lequel la coloration obtenue n'est pas une fonction croissante des numéros des sommets.

8) Écrire une fonction :

```
coloration : graphe -> int array
```

qui effectue l'algorithme de bonne coloration ci–dessus.

Nombre chromatique, fonctions h et k

- Soit g un graphe et p un entier strictement positif. On appelle **bonne p–coloration** de g une bonne coloration de g qui n'utilise que des couleurs appartenant à l'ensemble $[\![1, p]\!]$.
- $BC(g,p)$ désignera l'ensemble des bonnes p–colorations de g.
- $f_c(g,p)$ désignera le cardinal de $BC(g,p)$.
- Le **nombre chromatique** du graphe g désigne $\min\{p \in \mathbb{N}^* \mid f_c(g,p) > 0\}$: dit autrement, c'est le nombre minimum p de couleurs pour qu'il existe une bonne coloration de g avec p couleurs. Par exemple, le nombre chromatique du graphe $gex2$ vaut 2, et le nombre chromatique du graphe $gex1$ vaut 3.

- Le **premier voisin** d'un sommet s non isolé du graphe g est le voisin de s de numéro minimum. On le note $prem\_voisin(g, s)$. Par exemple, $prem\_voisin(gex1, 0)$ vaut 3.
- Étant donné un graphe g qui possède au moins une arête, on appelle **premier sommet non isolé** et on note $prem\_ni(g)$ le sommet non isolé de g de numéro minimum. Par exemple, $prem\_ni(gex1)$ vaut 0.

9) Écrire une fonction :

```
prem_voisin : graphe -> int -> int
```

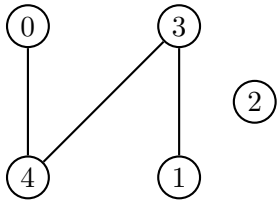
qui calcule le premier voisin d'un sommet dans un graphe. Cette fonction ne prévoira pas le cas où le sommet n'a aucun voisin.

10) Écrire une fonction :

```
prem_ni : graphe -> int
```

qui permet de calculer la fonction $prem\_ni$. Cette fonction ne prévoira pas le cas où g ne possède aucune arête.

11) Rappelons d'abord que dans la liste des arêtes d'un graphe g, il est possible qu'une même arête soit répétée.
Étant donné un graphe g possédant au moins une arête, on définit les sommets $s_1 = prem\_ni(g)$ et $s_2 = prem\_voisin(g, s_1)$.
On note $h(g)$ le graphe obtenu en supprimant toutes les arêtes entre s_1 et s_2.

Par exemple, $h(gex1)$ est la graphe 

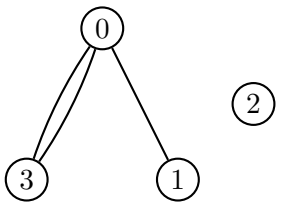
Écrire la fonction :

```
h : graphe -> graphe
```

Cette fonction ne prévoira pas le cas où g n'a aucune arête.

12) On considère un graphe g possédant au moins une arête. On définit s_1 et s_2 comme à la question précédente ; on peut remarquer que $s_2 > s_1$. On construit alors un graphe $k(g)$ de la façon décrite ci–dessous :

— On construit le graphe $h(g)$.
— dans $h(g)$ on superpose s_1 et s_2 ; plus précisément :
  — on considère successivement chaque arête de la liste des arêtes de $h(g)$; pour chacune d'entre elles, on renumérote ses extrémités :
    — une extrémité de valeur strictement inférieure à s_2 est inchangée ;
    — une extrémité de valeur s_2 prend la valeur s_1 ;
    — une extrémité de valeur strictement supérieure à s_2 est décrémentée de 1 ;
  — on diminue de 1 le nombre de sommets du graphe.

Par exemple, $k(gex1)$ est le graphe 

et est défini par : $n(k(gex1)) = 4$ et $e(k(gex1)) = [\{0, 3\}, \{3, 0\}, \{1, 0\}]$
Écrire la fonction :

```
k : graphe -> graphe
```

Cette fonction ne prévoira pas le cas où g ne possède aucune arête.

13) Lorsqu'un graphe g ne possède aucune arête, que vaut $f_c(g,p)$ en fonction de $n(g)$ et de p ?

14) On admettra le résultat suivant :

$$(E) \qquad f_c(g,p) = f_c(h(g),p) - f_c(k(g),p)$$

S'il vous reste du temps à la fin du TP, vous pouvez le démontrer. Indications : montrer que $BC(g,p) \subset BC(h(g),p)$, puis que $\text{card}(BC(k(g)),p) = \text{card}\left(BC(h(g),p) \setminus BC(g,p)\right)$.

Écrire une fonction récursive :

```
fc : graphe -> int -> int
```

qui utilise l'égalité (E) pour calculer $f_c(g,p)$. Le cas de base est le cas d'un graphe qui ne possède aucun arête.
Vérifier que l'appel `fc gex1 2` renvoie la valeur 0 ; et que l'appel `fc gex1 3` renvoie la valeur 36.

15) En déduire une fonction :

```
nombre_chromatique : graphe -> int
```

qui calcule le nombre chromatique d'un graphe.

Polynôme chromatique

Nous allons chercher à accélérer le calcul du nombre chromatique. Pour cela, en utilisant à nouveau l'égalité (E), et en raisonnant à la fois sur le nombre de sommets et le nombre d'arêtes d'un graphe, il est possible de montrer le résultat suivant : ≪ lorsque le graphe g est fixé, la fonction $p \mapsto f_c(g,p)$ est une fonction polynomiale de degré $n(g)$ à coefficients entiers ≫.

16) Démontrer ce résultat la fin du TP. Cette preuve est intéressante, donc prenez le temps de la rédiger. Vous pouvez ou bien procéder par récurrence sur le nombre de sommets et le nombre d'arêtes, ou bien en utilisant une relation d'ordre bien fondée sur $\mathbb{N}^2$.

17) Désormais un polynôme de degré au plus n de la forme $P = \sum_{k=0}^{n} c_k X^k$ sera codé par le tableau $[\![\mathtt{c_0}, \mathtt{c_1}, \ldots, \mathtt{c_n}]\!]$. Écrire une fonction :

```
difference : int array -> int array -> int array
```

qui permet de calculer la différence de deux polynômes. On supposera que le degré du premier polynôme est toujours strictement supérieur au degré du second polynôme.

18) En déduire une fonction :

```
polynome_chromatique : graphe -> int array
```

telle que, si g est un graphe, alors l'appel `polynome_chromatique g` permet de calculer le tableau des coefficients de la fonction polynomiale $p \mapsto f_c(g,p)$.

19) Écrire une fonction :

```
evalue_polynome : int array -> int -> int
```

telle que si `p` est un tableau d'entiers qui code le polynôme p, alors `evalue_polynome p x` retourne l'entier $p(x)$. La complexité de l'algorithme utilisé sera nécessairement du même ordre de grandeur que le degré du polynôme considéré.

20) En déduire une fonction :

```
nombre_chromatique_bis : graphe -> int
```

qui calcule le nombre chromatique d'un graphe, en utilisant le polynôme chromatique.