**Exercice 1. 3-COLOR**

Soit $G = (V, E)$ un graphe non orienté. On appelle $k$-**coloration** de $G$ une fonction $c : V \longrightarrow \{1, 2, \dots, k\}$ telle que pour tout arc $(u, v) \in E$, on a $c(u) \neq c(v)$.
On considère le problème suivant :

**3-COLOR**

Entrée : un graphe $G$ non orienté
Sortie : un booléen indiquant si $G$ est 3-colorable

1. Montrer que 3-COLOR appartient à la classe NP.
2. Écrire une fonction OCaml `check_3color` qui prend en entrée un graphe $G$ (représenté par liste d'adjacence) et une coloration $c$ (un tableau) et qui renvoie un booléen indiquant si $c$ est une coloration de $G$.
3. Donner une réduction de 3-SAT à 3-COLOR.

Dans la suite, on veut trouver une réduction de 3-COLOR à 3-SAT.
On considère une formule $\varphi$ de 3-SAT de variables $x_1, ..., x_n$. On veut construire un graphe $G$ qui soit 3-colorable si et seulement si $\varphi$ est satisfiable.
On commence par ajouter, dans $G$, $n$ sommets (encore appelés $x_1$, ..., $x_n$ par abus de notation) correspondant à $x_1$, ..., $x_n$, $n$ sommets correspondant à $\neg x_1$, ..., $\neg x_n$ et 3 sommets $T, F, B$ reliés 2 à 2.
Dans un 3-coloriage de $G$, $V$ et $F$ doivent être de couleurs différentes. Chaque variable $x_i$ sera considérée comme fausse si le sommet correspondant est de la même couleur que $F$ et vraie s'il est de la même couleur que $T$.

4. Expliquer comment ajouter des arêtes à $G$ pour que chaque variable soit vraie ou fausse (c'est-à-dire coloriée avec la même couleur que $F$ ou la même couleur que $T$).
5. Dessiner un graphe (un *gadget*) avec (au moins) 3 sommets $l_1$, $l_2$, $s$ que l'on pourrait ajouter à $G$ tels que :
   - Si $e_1$ et $e_2$ sont de la même couleur que $F$, alors $s$ doit être de même couleur que $F$.
   - Si $e_1$ ou $e_2$ est de la même couleur que $T$, alors il existe un coloriage de $G$ où $s$ est de la même couleur que $T$.
6. Soit $l_1 \vee l_2 \vee l_3$ une clause de $\varphi$. Expliquer comment ajouter un gadget à $G$ de façon à ce que $l_1 \vee l_2 \vee l_3$ soit vraie si et seulement si $G$ est 3-coloriable.
7. Montrer que 3-COLOR est NP-complet.
8. Appliquer la réduction ci-dessus à la formule $\varphi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_4)$.

Cependant, 2-COLOR est dans la classe P :

9. Écrire une fonction OCaml `2color` qui prend en entrée un graphe $G$ (représenté par liste d'adjacence) avec $n$ sommets et $p$ arêtes et qui renvoie un booléen indiquant si $G$ est 2-colorable, en $\text{O}(n + p)$.

**Exercice 2. Distance de Hamming**

Soit $\Sigma$ un alphabet. Si $u = u_1...u_n$ et $v = v_1...v_n$ sont deux mots de même longueur sur $\Sigma$, leur distance de Hamming est:

$$d(u, v) = |\{i \mid u_i \neq v_i\}|$$

1. Montrer que la distance de Hamming est une distance sur $\Sigma^*$.
2. Écrire une fonction `dist : 'a list -> 'a list -> int` calculant la distance de Hamming de deux mots de même longueur, sous forme de listes.

Étant donné un langage $L$ sur $\Sigma$, on définit son voisinage de Hamming $\mathcal{H}(L) = \{u \in \Sigma^* \mid \exists v \in L,\ d(u, v) = 1\}$.

3. Donner une expression rationnelle du voisinage de Hamming de $L(0^*1^*)$.
4. Définir par récurrence une fonction $H$ telle que, si $e$ est une expression rationnelle d'un langage $L$ sur $\Sigma = \{0, 1\}$, $H(e)$ est une expression rationnelle de $\mathcal{H}(L)$.
5. Écrire la fonction $H$ précédente en Caml.

On suppose disposer d'une structure impérative de dictionnaire en Caml de type `('a,'b) dict` avec les primitives suivantes :

| Primitive | Type | Description |
|---|---|---|
| `new` | `unit -> ('a,'b) dict` | Crée un nouveau dictionnaire vide |
| `add` | `('a,'b) dict -> 'a -> 'b -> unit` | `add d cl val` associe, dans le dictionnaire `d`, la clef `cl` à la valeur `val` |
| `find` | `('a,'b) dict -> 'a -> 'b` | `find d cl` lit, dans le dictionnaire `d`, la valeur associée à la clef `cl` |
| `keys` | `('a,'b) dict -> 'a list` | `keys d` renvoie la liste (dans un ordre arbitraire) des clefs du dictionnaire `d` |

`'a` est le type des clefs, et `'b` le type des valeurs associées aux clefs.

On suppose disposer d'une structure persistante d'ensemble d'entiers de type `set` avec les primitives suivantes :

| Primitive | Type | Description |
|---|---|---|
| `empty` | `set` | L'ensemble vide |
| `union` | `set -> set -> set` | L'union de 2 ensembles |
| `inter` | `set -> set -> set` | L'intersection de 2 ensembles |
| `card` | `set -> int` | Le cardinal d'un ensemble |
| `equal` | `set -> set -> bool` | Teste l'égalité de 2 ensembles |
| `mem` | `int -> set -> bool` | `mem i s` renvoie `true` si l'entier `i` appartient à l'ensemble `s` et `false` sinon |
| `singleton` | `int -> set` | `singleton i` renvoie l'ensemble à un élément ne contenant que `i` |

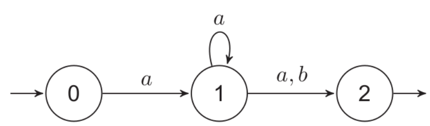
On représente un automate non-déterministe en Caml par le type suivant

```
type auto = {etats : set; init: set;
      trans: (int * char, set) dict ; final: set};;
```

Étant donné un automate `m`,

- `m.init` représente l'ensemble des états initiaux de `m` ;
- `m.final` l'ensemble de ses états finaux,
- `m.trans` sa fonction de transition qui à chaque couple `q,x` associe l'ensemble des états accessibles à partir de l'état `q` en lisant le caractère `x`.

Par exemple, considérons l'automate $\mathcal{M}_1$ suivant :




Si `m1` représente l'automate $\mathcal{M}_1$ en Caml alors `m1.final` est l'ensemble $\{2\}$ et `find m1.trans (1,`a`)` est l'ensemble $\{1;2\}$.

Un automate déterministe est un automate non-déterministe ayant un unique état initial et tel que pour tout état $q$ et toute lettre $a$, en lisant $a$ à partir de l'état $q$ on peut aller dans au plus un état.

1. L'automate $\mathcal{M}_1$ est-il déterministe ?
2. Écrire une fonction `max_card : ('a, set) dict -> int` qui étant donné un dictionnaire d'ensembles, renvoie le cardinal maximal des ensembles stockés dans le dictionnaire.
3. Écrire une fonction `est_deterministe : auto -> bool` qui renvoie `true` si l'automate donné en argument est déterministe et `false` sinon.
4. Considérons le code incomplet suivant :

```
let etats_suivants m s x =
let rec parcours_clefs clefs acc = match clefs with
        | [] -> acc
        | (etat,y)::r -> if ................
                then ................
                else ................
in parcours_clefs (keys m.trans) empty;;
```

Compléter le code de sorte que `etats_suivants m s x` renvoie l'ensemble des états accessibles à partir d'un état de l'ensemble `s` en lisant `x` dans l'automate `m`.

5. Écrire une fonction `reconnu : auto -> string -> bool` qui, étant donné un automate et une chaîne de caractères, renvoie `true` si l'automate reconnaît la chaîne et `false` sinon.
6. Si au lieu d'une structure impérative de dictionnaire nous avions une structure persistante de dictionnaire, quel serait le type de la primitive `add` ? Si nous avions une structure impérative d'ensemble d'entiers et non pas une structure persistante, pourquoi le type de `empty` devrait-il être changé ?