Dans ce sujet nous considérons un automate sur l'alphabet $A = \{0, 1\}$. Les états de l'automate sont codés par les entiers compris entre 0 et $n-1$ où n est le nombre d'états. Une transition est un triplet (p, x, q) où p désigne l'entier qui code l'origine, x l'entier qui code l'étiquette et q l'entier qui code l'extrémité. L'ensemble des transitions est appelé la *relation* de l'automate. L'ensemble des états initiaux et l'ensemble des états d'acceptation sont codés par des tableaux de booléens : si `i` est le tableau qui code les états initiaux, pour tout entier `k` $\in \{0, \ldots, n-1\}$, le booléen `i.(k)` vaut `true` si et seulement si `k` code un état initial. On prend le même principe pour le codage des états d'acceptation. Nous obtenons le codage suivant :

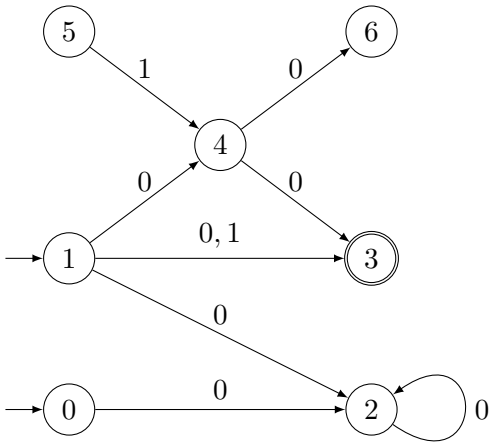
```
type etat = int;;
type etats = bool array;;
type transition = etat * int * etat;;
type automate = {taille : int ; relation : transition list ;
                 initiaux : etats ; finals : etats} ;;
type mot = int list;;
```

Note : les automates considérés seront en général *non déterministes.*

On pourra librement utiliser les fonctions suivantes :

- `Array.make : int -> 'a -> 'a array` telle que l'appel `Array.make n x` crée un tableau de taille `n` dont les éléments sont tous `x`.
- `Array.copy : 'a array -> 'a array` qui permet de dupliquer un tableau.

1. Coder l'automate suivant qui servira à effectuer les tests :



2. La lecture d'un mot pour un automate non déterministe (Q, A, E, I, T) se fait de façon suivante : à chaque étape de lecture on garde en mémoire l'ensemble des états où il est possible que l'on soit. Supposons que l'ensemble des états où il est possible de se trouver soit $\{q_1, \ldots, q_k\}$, alors la lecture d'un caractère c permet d'arriver à l'ensemble des états possibles

$$\{q'_1, \ldots, q'_l\} = \left\{ q \in Q \mid \exists p \in \{q_1, \ldots, q_k\}, (p, c, q) \in E \right\}$$

On part de l'ensemble des états initiaux. Un mot est reconnu par l'automate si et seulement si l'ensemble des états possibles auquel on arrive après la lecture de chaque lettre contient un état terminal.

Écrire une fonction

```
lecture_lettre : automate -> int -> etats -> etats
```

telle que pour un automate `a`, si on se trouve dans un ensemble d'états possibles `e` (codé par un tableau de booléens), la lecture de l'entier `c` permet d'arriver à l'ensemble d'états possibles calculés par `lecture_lettre a c e` (ce résultat est encore codé dans un ensemble de booléens).

3. Écrire une fonction :

```
contient_terminal : automate -> etats  -> bool
```

telle que `contient_terminal a e` permet de savoir s'il existe ou non parmi les états de `e` un état d'acceptation de l'automate `a`.

4. En déduire une fonction

```
lecture :  automate -> mot -> bool
```

qui permet de savoir si un mot est reconnu par un automate ou non.

5. Un état d'un automate est dit *accessible* s'il est possible d'y arriver en lisant un mot depuis un état initial : dit autrement, un état q d'un automate (Q, A, E, I, T) est dit accessible s'il existe un état initial i, des états $q_2, \ldots, q_n = q$ et des symboles de l'alphabet $c_1, \ldots, c_n \in A$ tels qu'on ait la séquence de transitions :

$$i \xrightarrow{c_1} q_2 \xrightarrow{c_2} \ldots q_{n-1} \xrightarrow{c_n} q_n = q$$

Le cas $n = 0$ est autorisé : on considère que tout état initial est accessible, par lecture du mot vide.

Il est clair que lors de la lecture d'un mot par un automate, seuls les états accessibles ont un intérêt.

Pour calculer la liste des états accessibles on procède ainsi : on mémorise deux tableaux de booléens, le tableau des états accessibles et le tableau des états en cours d'exploration.

Au départ les deux tableaux sont des copies du tableau des états initiaux.

Les états en cours d'exploration correspondent aux états accessibles dont on n'a pas encore exploré les successeurs.

En s'inspirant de ce principe, écrire une fonction :

```
accessibles : automate -> etats
```

qui calcule la liste des états accessibles de l'automate.

6. Un état d'un automate est dit *co–accessible* s'il est possible en partant de lui d'arriver à un état d'acceptation : dit autrement, un état q d'un automate (Q, A, E, I, T) est dit accessible s'il existe un état d'acceptation t, des états $q_1 = q, q_2, \ldots, q_{n-1}$ et des symboles de l'alphabet $c_1, \ldots, c_n \in A$ tels qu'on ait la séquence de transitions :

$$q = q_1 \xrightarrow{c_1} q_2 \xrightarrow{c_2} \ldots q_{n-1} \xrightarrow{c_n} t$$

   Le cas $n = 0$ est autorisé : on considère que tout état d'acceptation est co–accessible, par lecture du mot vide.

   Il est clair que lors de la lecture d'un mot, pour savoir si ce mot est reconnaissable ou non, seuls les états co–accessibles ont un intérêt : si à un moment on arrive à un état qui n'est pas co–accessible alors on ne pourra jamais atteindre un état d'acceptation.

   Pour déterminer les états co–accessibles on peut utiliser la notion de *miroir* d'un automate : si $\mathscr{A} = (Q, A, E, I, T)$ est un automate, son automate–miroir est $mir(\mathscr{A}) = (Q, A, E', T, I)$ où $E' = \{(q, c, q') \,|\, (q', c, q) \in E\}$ (c'est–à–dire qu'on a interverti l'origine et l'extrémité de chaque transition). Remarquer par ailleurs que les états initiaux de $mir(\mathscr{A})$ sont les états d'acceptation de $\mathscr{A}$ et réciproquement.

   Écrire une fonction :

   ```
   miroir : automate -> automate
   ```

   qui calcule le miroir d'un automate.

7. En utilisant qu'un état de $\mathscr{A}$ est co–accessible si et seulement s'il est accessible dans $mir(\mathscr{A})$, écrire une fonction :

   ```
   coaccessibles : automate -> etats
   ```

   qui calcule la liste des états co–accessibles d'un automate.

8. Écrire une fonction:

   ```
   accessibles_et_co : automate -> etats
   ```

   qui permet de calculer les états à la fois accessibles et co–accessibles.

9. Pour émonder un automate, on supprime tous les états qui ne sont pas à la fois accessibles et co–accessibles. Attention : avec notre codage, si n est le nombre d'états d'un automate, tous les états de 0 à $n - 1$ doivent être utilisés. Il faudra donc renuméroter les états, et en fonction de cette renumérotation redéfinir les transitions, les états initiaux et les états d'acceptation.

   Écrire une fonction :

   ```
   emonde : automate -> automate
   ```

   qui permet de calculer un automate dans lequel on n'a gardé que les états à la fois accessibles et co–accessibles.

10. Écrire une fonction `deterministe : automate -> bool` telle que `deterministe auto` renvoie `true` si et seulement si l'automate `auto` est déterministe.