Dans tout le TP, on considère $G = (S, A)$ un graphe pondéré connexe dont les sommets sont étiquetés par $0, 1, \ldots, n-1$ et dont les arêtes sont pondérées par des poids strictement positifs. En Caml, un graphe est représenté par matrice d'adjacence :

```
type graphe = int array array;;
```

Exercice 1. Implémentation de l'algorithme de Kruskal

Lors de l'exécution de l'algorithme de Kruskal sur G, on définit un graphe $T = (S, \varnothing)$ auquel on ajoute des arêtes de A jusqu'à obtenir un arbre couvrant. Chaque arête ajoutée doit relier deux sommets n'appartenant pas à la même composante connexe dans T. On aura donc besoin de tester si deux sommets se trouvent dans la même composante connexe. Pour cela, on définit un nouveau type `compo_connexes` muni de trois opérations :

- `creer: int -> compo_connexe`
  Un appel à « `creer n` » renvoie un objet « `cc: compo_connexe` » représentant les composantes connexes d'un graphe avec `n` sommets, mais aucune arête.
- `meme_compo: compo_connexe -> int -> int -> bool`
  Un appel à « `meme_compo cc u v` » indique si les sommets `u` et `v` sont dans la même composante connexe.
- `fusion: compo_connexe -> int -> int -> unit`
  Un appel à « `fusion cc u v` » fusionne les composantes connexes associées à `u` et `v`. La fonction déclenche une erreur si `u` et `v` sont déjà dans la même composante connexe.

Le programme ci-contre est un exemple d'utilisation de ces fonctions. Le tableau ci-dessous donne en fonction du couple `(u,v)`, les valeurs des appels à « `meme_compo cc u v` » :

```
let cc = creer 4 in
    fusion cc 0 3;
    fusion cc 2 1;
    fusion cc 1 3;;
```

`(u,v)`	Après ligne 1	Après ligne 2	Après ligne 3	Après ligne 4
`(0,0)`, `(1,1)`, `(2,2)` ou `(3,3)`	`true`	`true`	`true`	`true`
`(0,1)` ou `(1,0)`	`false`	`false`	`false`	`true`
`(0,2)` ou `(2,0)`	`false`	`false`	`false`	`true`
`(0,3)` ou `(3,0)`	`false`	`true`	`true`	`true`
`(1,2)` ou `(2,1)`	`false`	`false`	`true`	`true`
`(1,3)` ou `(3,1)`	`false`	`false`	`false`	`true`
`(2,3)` ou `(3,2)`	`false`	`false`	`false`	`true`

1. Proposer une implémentation pour le type `compo_connexe` et les trois fonctions associées.

Pour écrire l'algorithme de Kruskal en Caml, il va falloir dans un premier temps trier les arêtes en fonction de leur poids. Une arête est représentée par une variable « `a: arete` » telle que `a.s1` et `a.s2` sont les deux sommets de l'arête et `a.poids` est son poids.

```
type arete = {
    s1: int;
    s2: int;
    poids: int;
  };;
```

Pour trier une liste « `li: 'a list` », on utilisera la fonction :

`List.sort: ('a -> 'a -> int) -> 'a list -> 'a list`

Le premier argument de `List.sort` est une fonction `comp` telle que « `comp x y` » vaut 0 si `x` et `y` sont considérés égaux, est strictement positif si `x` est strictement plus grand que `y` et est strictement négatif si `x` est strictement plus petit que `y`. Dans notre cas, on doit donc écrire une fonction « `comp: arete -> arete -> int` » telle qu'un appel à « `comp a1 a2` » s'évalue en :

- `0` si les deux arêtes ont le même poids.
- Un entier strictement positif si `a1` a un poids strictement supérieur à `a2`.
- Un entier strictement négatif sinon.

2. (a) Écrire une fonction « `liste_aretes : graphe -> arete list` » qui renvoie la liste des arêtes du graphe (sans doublons).
   (b) Écrire la fonction « `comp : arete -> arete -> int` » comme décrit ci-dessus.
   (c) En déduire une fonction « `kruskal: graphe -> graphe` » qui prend en entrée un graphe connexe et renvoie un arbre couvrant de poids minimal en utilisant l'algorithme de Kruskal.

Exercice 2. Algorithme de Prim

Comme l'algorithme de Kruskal, l'algorithme de Prim permet de construire un arbre couvrant de poids minimal d'un graphe connexe $G = (S, A)$. En voici le principe :

- Soit r un sommet quelconque de G. Soit T le graphe ne contenant que le sommet r, c'est à dire que $T = (\{r\}, \varnothing)$.
- Soit S' l'ensemble des sommets de T. Pour chaque $s \in S \setminus S'$, on pose :

$$D(s) = \Big\{\omega(s,t) : t \in S' \text{ et } \{s,t\} \text{ est une arête de } G\Big\}.$$

où $\omega(s,t)$ est le poids de l'arête $\{s,t\}$. On définit alors :

$$d(s) = \begin{cases} +\infty & \text{si } D(s) = \varnothing, \\ \min(D(s)) & \text{sinon.} \end{cases}$$

Tant que $S' \neq S$:

- Soit $s_0 \in S \setminus S'$ tel que $d(s_0)$ est minimum ($d(s_0) \neq +\infty$ car G est connexe). Soit $t \in S'$ l'un des sommets tels que $\omega(s_0, t) = d(s_0)$.
- On ajoute le sommet s_0 et l'arête $\{s_0, t\}$ au graphe T.

Implémenter l'algorithme de Prim en OCaml (si besoin, vous pouvez lire les indications ci-dessous).

Indications (essayez de résoudre l'exercice sans lire ce qui suit). Tout au long de l'algorithme, on va manipuler une variable `(fp: file_priorite)` :

```
type voisin_plus_proche =
  | None
  | Some of int;;
```

```
type file_prio =
  (int * voisin_plus_proche) list ref;;
```

À chaque étape, la liste `fp` est de taille card$(S \setminus S')$. De plus, pour chaque sommet `s` de $S \setminus S'$, la file `fp` contient un couple `(s,t0)` où :

- Si `s` n'a pas de voisin dans S' alors `t0` vaut `None`.
- Sinon, `t0` vaut `(Some t)` où `t` est le sommet de S' qui minimise ω`(s, t)`.

À chaque étape de l'algorithme, il suffit de récupérer l'élément de `fp` correspondant à un poids minimal. Cet élément correspond à une arête $\{s_0, t\}$, qui est l'arête à ajouter à T.

Exercice 3. Preuve de la validité de l'algorithme de Kruskal

Le but de cet exercice est de montrer que l'algorithme de Kruskal appliqué à un graphe $G = (S, A)$ renvoie bien un arbre couvrant de poids minimal. Dans un premier temps, on aura besoin de montrer quelques lemmes concernant les graphes.

Lemme 1. *Soit $G = (S, A)$ un graphe connexe, alors il existe un arbre couvrant pour G.*

Définition 2. Une chaîne est dite ***élémentaire*** si elle ne passe pas deux fois par le même sommet.

Lemme 3. *Soit $G = (S, A)$ un graphe non orienté et u, v deux sommets. S'il existe une chaîne de u à v alors il existe une chaîne élémentaire de u à v.*

Lemme 4. *Soit $T = (S, A)$ un graphe et u, v deux sommets. On suppose que $\{u, v\}$ n'est pas une arête de T, mais qu'il existe une chaîne de u à v dans T. Soit G le graphe obtenu en ajoutant l'arête $\{u, v\}$ à T. Alors G contient un cycle passant par l'arête $\{u, v\}$.*

Définition 5. Soit G un graphe et u un sommet. On note $CC_G(u)$ la composante connexe de u dans G, c'est à dire :

$$CC_G(u) = \Big\{ v \in S : \text{il existe une chaîne de } u \text{ à } v \text{ dans } G \Big\}$$

Lemme 6. *Soient :*

- *$G_1 = (S, A_1)$ un graphe connexe et $\{u_1, v_1\}$ une arête de G_1.*
- *$G_2 = (S, A_2)$ le graphe G_1 privé de l'arête $\{u_1, v_1\}$.*
- *$u_2 \in CC_{G_2}(u_1)$ et $v_2 \in CC_{G_2}(v_1)$.*
- *$G_3 = (S, A_3)$ le graphe G_2 auquel on a ajouté l'arête $\{u_2, v_2\}$. Notez que $G_3 = G_2$ si $\{u_2, v_2\}$ est déjà une arête de G_2.*

Alors G_3 est connexe.

Lemme 7. *Soit $G_1 = (S, A_1)$ un graphe sans cycle et u_1, v_1 deux sommets appartenant à deux composantes connexes différentes de G_1. Soit $G_2 = (S, A_2)$ le graphe G_1 auquel on a ajouté l'arête $\{u_1, v_1\}$. Alors G_2 ne contient pas de cycle.*

1. Montrer les cinq lemmes ci-dessus.

Pour montrer la correction de l'algorithme de Kruskal, on utilise un invariant de boucle. On note $T_0, T_1, \ldots, T_k$ les graphes construits par l'algorithme. Ainsi, $T_0 = (S, \varnothing)$; pour tout i, T_{i+1} est le graphe T_i auquel on a ajouté une arête ; et T_k est l'arbre couvrant renvoyé par l'algorithme. Pour chaque i, on note A_i l'ensemble de arêtes de T_i. On pose :

$$(\mathcal{P}_i) : \text{Il existe } T = (S, B) \text{ un arbre couvrant de poids minimal de } G \text{ tel que } A_i \subset B$$

2. (a) Montrer $(\mathcal{P}_0)$.
   (b) Soit $i \in [\![0; k-1]\!]$. Montrer que $(\mathcal{P}_i)$ implique $(\mathcal{P}_{i+1})$.
   (c) En déduire la validité de l'algorithme de Kruskal.