TD 8

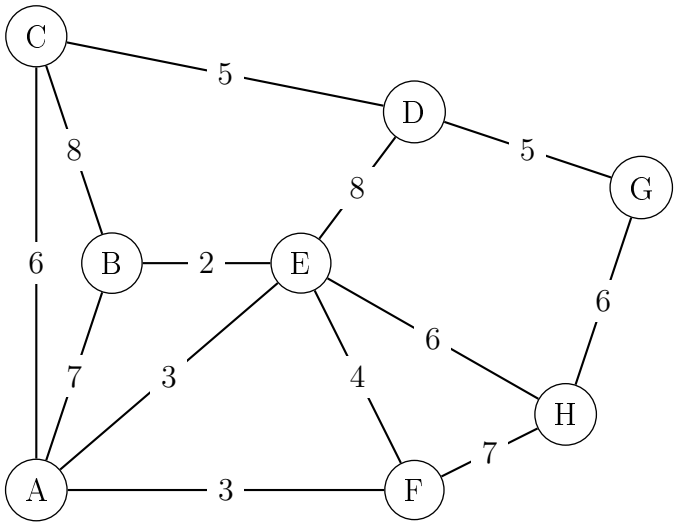
Arbres couvrants

L'étude porte sur des graphes non orientés connexes et pondérés à poids positifs.
On rappelle qu'un arbre est un graphe connexe sans cycle.
Dans ce TD, le graphe `g` sera représenté par listes d'adjacence pondérées : `g.(i)` contient la liste des sommets adjacents à `i` dans `g` avec le poids de l'arête associée :

```
type graphe =(int*int) list array ;;
```

Un arbre couvrant du graphe $G = (V, E)$ est un arbre $A = (V, E')$ connexe tel que $E' \subset E$.

L'objectif est de trouver un arbre couvrant minimal (soit, de poids minimum) de G.
On considère le graphe pondéré suivant :



La liste d'adjacence est donc :

```
let graphe_exemple =
  [| [(1,7) ; (2,6) ;(4,3) ; (5,3)] ;
  [ (0,7 ) ; (2,8) ; (4 , 2)] ;
  [ ( 0,6) ; (1 , 8 ) ; ( 3 , 5)] ;
  [(2,5) ; ( 4, 8) ; (6 , 5)] ;
  [(0,3);(1,2) ; (3,8) ; ( 5,4) ; ( 7,6)] ;
  [(0,3) ; (4,4) ; ( 7,7)] ;
  [(3,5) ; (7,6)];
  [(4,6) ; ( 5 , 7) ; (6,6)] |] ;;
```

1 Algorithme de Kruskal

L'algorithme de Kruskal est un algorithme de recherche d'arbre recouvrant de poids minimum. L'algorithme considère toutes les arêtes du graphe par poids croissant (en pratique, on trie d'abord les arêtes du graphe par poids croissant) et pour chacune d'elles, il la sélectionne si elle ne crée pas un cycle.

Appliquer l'algorithme sur le graphe exemple.

2 Gestion des composantes connexes dans la construction de l'arbre

Pour construire l'arbre couvrant, nous allons partir du sous-graphe sans arête $A = (V, \varnothing)$. Chaque arête ajoutée doit relier deux sommets qui n'appartiennent pas à la même composante connexe dans A .
Chaque composante connexe sera mis en mémoire dans un tableau dont les éléments seront un représentant de la classe.

On utilisera le type :

```
type composante_connexe =int array ;;
```

1. Écrire la fonction `creer_cc` qui prend en argument un entier n (le nombre de sommets) et qui renvoie le tableau des composantes connexes pour un arbre sans arête.

   ```
   val creer_cc : int -> composante_connexe
   ```

   Exemple :

   ```
   creer_cc ( 12);;
   - : composante_connexe = [|0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11|]
   ```

2. Écrire la fonction `classe` qui prend en argument le tableau de la composante connexe et deux entier i et j et renvoie le booléen vérifiant si les sommets sont dans la même composante connexe.

   ```
   classe : composante_connexe -> int -> int -> bool
   ```

3. Écrire la fonction `fusion` qui prend en argument le tableau de la composante connexe et deux entier i et j et fusionne les composantes connexes. On pourra supposer que les composantes connexes étaient distinctes. La fonction ne renvoie rien.

   ```
   fusion : composante_connexe -> int -> int -> unit
   ```

   Exemple :

   ```
   fusion [|0;2;2;1;1;6;1;2;4;4;0; 6 ; 1|] 2 5;;
   - : composante_connexe = [|0; 2; 2; 1; 1; 2; 1; 2; 4; 4; 0; 2; 1|]
   ```

3 Trie des arêtes

La première étape consiste au tri des arêtes dans l'ordre croissant. L'idée ici est d'utiliser le tri par insertion pour construire la liste des arêtes.

1. Écrire la fonction `insertion` qui prend un argument un sommet, un couple (sommet,poids) et une liste triée de couples d'arêtes avec leur poids. La fonction retourne la liste avec la nouvelle arête à la bonne place. Si l'arête existe déjà, la fonction renvoie la liste initiale.

   ```
   insertion :
     'a -> 'a * 'b -> (('a * 'a) * 'b) list -> (('a * 'a) * 'b) list
   ```

   Exemple :

```
insertion 3 (2,5) [((2,0),1);((5,2),1); ((1,4),4) ; ((1,3),6)];;
- : ((int * int) * int) list =
[((2, 0), 1); ((5, 2), 1); ((1, 4), 4); ((3, 2), 5); ((1, 3), 6)]
```

2. Écrire la fonction `liste_arete_trie` qui prend un argument un graphe et renvoie la liste des arête ainsi que leur poids.

   ```
   liste_arete_trie : graphe -> ((int * int) * int) list
   ```

   Exemple :

   ```
   liste_arete_trie graphe_exemple ;;
   - : ((int * int) * int) list =
   [((1, 4), 2); ((5, 0), 3); ((4, 5), 4); ((2, 3), 4); ((3, 6), 5);
    ((0, 3), 5); ((7, 6), 6); ((7, 4), 6); ((6, 7), 6); ((4, 7), 6);
    ((0, 2), 6); ((5, 7), 7); ((0, 1), 7); ((3, 4), 8); ((1, 2), 8)]
   ```

4 Implémentation de l'algorithme de Kruskal

1. A l'aide des fonctions précédentes, écrire la fonction `kruskal` qui prend en argument un graphe et renvoie la matrice d'adjacence de l'arbre couvrant de poids minimal.

   ```
   kruskal : graphe -> int array array
   ```

2. Modifier la fonction précédente pour avoir en plus le poids minimal.

   ```
   kruskal_poids : graphe -> int array array * int
   ```