Préliminaires. Télécharger et dézipper le fichier `tp08.zip` sur la page web du cours.
Il contient les fichiers suivants :

- un fichier `tasMin.ml` déjà rempli, contenant l'implémentation d'une structure de priorité-min via des tas-min ;
- un fichier `unionFind.ml` destiné à implémenter une structure Union-Find (cf. partie 1) ;
- des fichiers `tasMin.mli` et `unionFind.mli` contenant les types des fonctions définis dans les fichiers `.ml` correspondants (c'est l'équivalent pour OCaml des fichiers `.h` de C) ;
- un fichier `tp08.ml` dans lequel nous implémenterons l'algorithme de Kruskal (cf. partie 2) et l'algorithme de recherche d'un couplage maximum dans un graphe biparti(cf. partie 3).

1 Structure Union-Find

Pour cette partie, ouvrir le fichier `unionFind.ml` dans votre éditeur de texte.
Notre objectif est d'implémenter les 3 fonctions suivantes :

unionFind.mli

```
val create : int -> uf
val find : uf -> int -> int
val union : uf -> int -> int -> unit
```

On rappelle le principe de la structure Union-Find : on souhaite représenter de manière efficace une partition d'un ensemble de la forme $[\![0, n-1]\!]$, en utilisant des arbres tels que :

- i et j sont dans la même classe de la partition si et seulement s'ils sont dans le même arbre de la structure ;
- la racine d'un arbre de la structure est appelé le **représentant** de la classe ;
- pour **tester** si i et j sont dans la même classe, on teste s'ils ont le même représentant ;
- pour **fusionner** deux classes, on branche le représentant d'une des classes sous le représentant de l'autre classe.

type uf

```
type uf = { link: int array; rank: int array }
```

Pour représenter une telle structure, on utilise le type `uf` ci-dessus, où :

- `link` est un tableau de taille n tel que `link.(i)` contient le parent de i dans son arbre ;
- on a `link.(i) = i` si et seulement si i est le représentant de sa classe ;
- si i est le représentant de sa classe, `rank.(i)` contient une surapproximation de la hauteur de l'arbre de i.

1. Implémenter la fonction `create : int -> uf` telle que `create n` renvoie une structure Union-Find représentant la partition de $[\![0, n-1]\!]$ en n singletons.

Compression de chemins. On rappelle l'optimisation suivante à implémenter dans la fonction `find` : `find` doit être une fonction récursive, et doit, lorsqu'on dépile ses appels récursifs, mettre à jour le tableau `link` pour que tous les nœuds rencontrés lors de la recherche du représentant aient pour parent ce représentant.

2. Implémenter la fonction `find : uf -> int -> int` prenant en arguments une structure Union-Find et un entier, et renvoyant le représentant de cet entier. Votre fonction modifiera également la structure Union-Find par effets de bords pour **compresser** le chemin rencontré.

Union pondérée. On rappelle l'optimisation suivante à implémenter dans la fonction `union` : plutôt que de choisir arbitrairement quel représentant doit devenir le fils de l'autre représentant, on utilise les valeurs du tableau `rank` pour faire le choix qui va minimiser la hauteur de l'arbre obtenue : c'est le représentant ayant le plus petit rang qui devient le fils de l'autre représentant. En cas d'égalité des rangs des deux représentants, on choisit arbitrairement et on met à jour le rang du représentant se trouvant à la racine du nouvel arbre obtenu (la hauteur de l'arbre a été augmentée de 1).

3. Implémenter la fonction `union : uf -> int -> int -> unit` prenant en arguments une structure Union-Find et deux entiers, et fusionnant les classes des deux entiers selon la stratégie précédente en modifiant la structure Union-Find par effets de bords.

Compilation. Pour les parties suivantes, ouvrir le fichier `tp08.ml` avec votre éditeur de texte. Dans ces parties, nous allons devoir utiliser les fonctions présentes dans les fichiers `tasMin.ml` et `unionFind.ml`. Pour cela, nous allons compiler les fichiers en question. Dans un terminal :

- Taper la commande : `ocamlc tasMin.mli`
  ↪ Cette commande a créé un nouveau fichier `tasMin.cmi` dans votre dossier.
- Taper la commande : `ocamlc unionFind.mli`
  ↪ Cette commande a créé un nouveau fichier `unionFind.cmi` dans votre dossier.

Ces fichiers vont permettre à Merlin (l'extension VSCode qui analyse votre code et souligne vos erreurs) de connaître les fonctions des modules `TasMin` et `UnionFind`. Depuis un fichier extérieur aux modules, ces fonctions s'appellent :

Module TasMin
```
TasMin.creer : 'a * 'b -> ('a,'b) TasMin.tasmin
TasMin.taille : ('a,'b) TasMin.tasmin -> int
TasMin.est_vide : ('a,'b) TasMin.tasmin -> bool
TasMin.push : ('a,'b) TasMin.tasmin -> 'a -> 'b -> unit
TasMin.peek : ('a,'b) TasMin.tasmin -> 'a
TasMin.pop : ('a,'b) TasMin.tasmin -> 'a
```

Module UnionFind
```
UnionFind.create : int -> UnionFind.uf
UnionFind.find : UnionFind.uf -> int -> int
UnionFind.union : UnionFind.uf -> int -> int -> unit
```

Mais pour que le toplevel OCaml puisse utiliser le code de vos fonctions, il faut également compiler les fichiers `tasMin.ml` et `unionFind.ml`. Dans le terminal, taper les commandes suivantes :

- `ocamlc tasMin.ml`
- `ocamlc unionFind.ml`

Cela va créer deux nouveaux fichiers : `tasMin.cmo` et `unionFind.cmo` contenant le bytecode de vos fichiers `.ml`. Pour les importer, taper les commandes suivantes dans le toplevel OCaml (ne pas oublier le `#`) :

Toplevel OCaml
```
# #load "tasMin.cmo" ;;
# #load "unionFind.cmo" ;;
```

Si le toplevel OCaml ne trouve pas vos fichiers `.cmo`, le fermer (`exit 0;;`) et en ouvrir un nouveau.

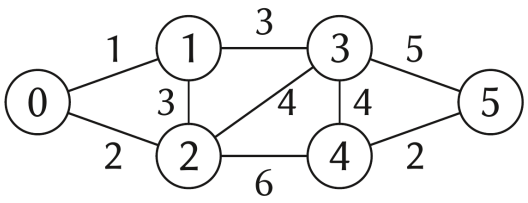
Désormais, vous pouvez utiliser les fonctions des modules `TasMin` et `UnionFind` sans problème dans le fichier `tp08.ml`.

2 Algorithme de Kruskal

Dans cette section, on travaille avec des graphes non orientés non pondérés, représentés par des listes d'adjacence : si `g : (int * 'a) list array` est un tel graphe, `g.(i)` est une liste de couples de la forme (j, ω) tels que $i \xrightarrow{\omega} j$ est une arête du graphe.

Remarque. En général, le type `'a` qui représente les poids et le type `float`.

Exemple. La représentation du graphe suivant est fourni dans le fichier `tp08.ml` afin de pouvoir tester vos programmes :



4. Implémenter la fonction `edges : (int * 'a) list array -> ((int * int) * 'a) list` prenant en argument un graphe non orienté pondéré représenté par listes d'adjacence et renvoyant la liste de ses arêtes.
   Remarque : le graphe étant non orienté, on ne gardera dans la liste que les arêtes $i \xrightarrow{\omega} j$ telles que $i < j$ (pour ne pas les avoir en double).

Exemple

```
# edges g ;;
- : ((int * int) * float) list =
[((4, 5), 2.); ((3, 5), 5.); ((3, 4), 4.); ((2, 4), 6.); ((2, 3), 4.);
 ((1, 3), 3.); ((1, 2), 3.); ((0, 2), 2.); ((0, 1), 1.)]
```

On rappelle ici l'algorithme de Kruskal, utilisant une file de priorité-min pour calculer un arbre couvrant de poids minimal d'un graphe non orienté pondéré :

Algorithme 1 : Algorithme de Kruskal

Données : Un graphe non orienté pondéré $G = (S, A, \omega)$
Résultat : Un arbre couvrant de G de poids minimal
$U \leftarrow$ une structure **union-find** pour les sommets $S = [\![0, n-1]\!]$;
$Q \leftarrow$ une **file de priorité-min** ;
$T \leftarrow \{\}$;
pour $x - y \in A$ **faire**
  Ajouter $(x - y, \omega(x - y))$ dans Q ;
tant que *longueur*$(T) < n - 1$ **faire**
  $x - y \leftarrow$ élément de Q de poids minimal ;
  si *x et y ne sont pas dans la même classe pour U* **alors**
    $T \leftarrow T \cup \{x - y\}$;
    Fusionner dans U les classes de x et y ;
retourner T ;

5. Implémenter la fonction `kruskal : (int * float) list array -> (int * int) list` prenant en argument un graphe non orienté pondéré représenté par listes d'adjacence et renvoyant un arbre couvrant de poids minimal.

Exemple

```
# kruskal g ;;
- : (int * int) list = [(3, 4); (1, 3); (0, 2); (4, 5); (0, 1)]
```

3 Couplage maximum dans un graphe biparti

Dans cette partie, on travaille avec des graphes non orientés bipartis. Le but est de trouver un couplage maximum en cherchant des chemins augmentants, via l'algorithme suivant :

Algorithme 2 : Algorithme de couplage maximum

Données : Un graphe non orienté biparti $G = (S, A)$ avec $S = X \uplus Y$
Résultat : Un couplage maximum de G
$C \leftarrow \emptyset$;
pour $x \in X$ **faire**
  chercher un chemin augmentant depuis x (via un parcours en profondeur) ;
  si *un tel chemin a été trouvé* **alors**
    modifier C pour inverser les couplages le long de ce chemin ;
retourner C ;

6. Implémenter la fonction `couplage_maximum` de type `int list array -> (int -> bool) -> (int * int) list` prenant en arguments un graphe `g` représenté par listes d'adjacence et la fonction caractéristique `fX` de X, et renvoie un couplage maximum de `g`.
   On pourra organiser le code de la manière suivante :
   - utiliser un tableau `c` pour stocker les couplages (`c.(y) = x` si (x, y) fait partie du couplage ; `c.(y) = -1` sinon) ;
   - implémenter une fonction locale `augment` qui cherche un tel chemin augmentant, et renvoie `true` ou `false` indiquant si la recherche est un succès ;
   - en cas de réponse positive, on inversera les couplages le long du chemin lorsqu'on dépile les appels récursifs ;
   - une fois les recherches de chemins augmentants terminés, on construira la liste des couplages à partir du tableau `c`.

Exemple

```
# couplage_maximum g1 fX ;;
- : (int * int) list = [(2, 5); (4, 3); (0, 1)]
# couplage_maximum g2 fX ;;
- : (int * int) list = [(0, 7); (2, 5); (6, 3); (4, 1)]
```