# OPTION INFORMATIQUE
# Devoir surveillé 1 (2h)

*Toutes les fonctions sont à écrire en OCaml. On attachera le plus grand soin à leur lisibilité. On pourra toujours librement utiliser une fonction demandée à une question précédente, même si cette question n'a pas été traitée. Lorsqu'une question impose une contrainte sur le coût en temps d'une fonction, on ne demande pas de démontrer que cette contrainte est respectée.*

***Pour toute fonction programmée, on précisera le type de cette fonction.***

## 1 Représentation d'ensembles par des listes

Dans cette partie, on représente un ensemble fini par la liste **triée** (dans l'ordre croissant) **sans doublon** de ses éléments. On identifiera alors l'ensemble et la liste le représentant.

1. Écrire une fonction calculant le cardinal d'un ensemble.
2. Écrire une fonction renvoyant l'élément minimal d'un ensemble non vide.
3. Écrire une fonction renvoyant l'élément maximal d'un ensemble non vide.
4. Écrire une fonction testant l'appartenance d'un élément à un ensemble.
5. Écrire une fonction `comprehension` prenant en argument un prédicat $p$ et un ensemble $l$ et renvoyant l'ensemble des éléments de $l$ vérifiant $p$.
6. Écrire une fonction `insertion` prenant en argument un élément $x$ et un ensemble $l$ et renvoyant l'ensemble contenant $x$ et les éléments de $l$.
7. Écrire une fonction renvoyant l'union de deux ensembles. Chaque liste ne devra être parcourue qu'au plus une fois.
8. Écrire sur le même principe une fonction renvoyant l'intersection de deux ensembles, une fonction renvoyant la différence ensembliste de deux ensembles, et une fonction testant l'inclusion d'un ensemble dans un autre.
9. On cherche à écrire une fonction `images` renvoyant l'ensemble des images des éléments d'un ensemble par une fonction $f$. L'enjeu est de s'assurer que ces images vont bien apparaître dans l'ordre croissant et sans doublon dans le résultat.
   (a) Écrire une version de cette fonction `images` utilisant la fonction `insertion`.
   (b) Donner un exemple de fonction $f$ pour laquelle le calcul de la version précédente de `images` a un coût quadratique en la longueur de la liste.
   (c) Pour améliorer le coût en moyenne, on se propose d'utiliser un tri rapide. Écrire une fonction `repartition` prenant en argument une liste $l$ (non nécessairement triée ou sans doublon) et un pivot p, et renvoyant la liste des éléments de l strictement inférieurs au pivot et la liste des éléments strictement supérieurs au pivot.
   (d) En déduire une fonction récursive `tri_rapide` prenant en argument une liste non triée et renvoyant l'ensemble de ses éléments (sous forme de liste triée sans doublon)
   (e) Utiliser cette fonction `tri_rapide` pour obtenir une nouvelle version de la fonction `images`.
10. Écrire une fonction `produit` réalisant le produit cartésien de deux ensembles. On notera que l'ordre que Caml utilise sur les couples est l'ordre lexicographique : `produit [1; 2; 3] [4; 5]` doit renvoyer `[(1, 4); (1, 5); (2, 4); (2, 5); (3, 4); (3, 5)]`.
11. Écrire une fonction `parties` renvoyant l'ensemble des parties d'un ensemble. Là encore, on notera que Caml utilise l'ordre lexicographique sur les listes, ainsi `parties [1; 2; 3]` doit renvoyer `[[]; [1]; [1; 2]; [1; 2; 3]; [1; 3]; [2]; [2; 3]; [3]]`.

## 2 Parcours d'arbre

On définit en Ocaml, pour représenter des arbres avec à chaque noeud un nombre quelconque de fils, le type suivant :

```
type 'a arbre = V | N of 'a * 'a arbre list;;
```

1. Lister sans justification les noeuds dans le parcours postfixe de l'arbre
   `a = N (9, [N (1, []); N (2, [N (4, [N (1, []); N (3, [])])])])`
2. Écrire une fonction `hauteur` calculant la hauteur d'un tel arbre (on rappelle que par convention l'arbre vide est de hauteur $-1$).
3. Écrire une fonction `parcours_largeur` prenant en argument une procédure p et un arbre a, et appliquant p aux noeuds de a dans l'ordre d'un parcours en largeur (chaque niveau étant parcouru de gauche à droite).

## 3 Minimum exclus

L'enjeu de cet exercice est de calculer le minimum exclus (ou **mex**) d'un tableau d'entiers naturels, c'est-à-dire le plus petit entier naturel qui n'apparaît pas dans le tableau. Par exemple, le mex de `[|2; 1; 0; 5|]` est 3, le mex de `[|4;7|]` est 0.

1. Donner sans justification le mex de `[|3; 6; 0|]`
2. (a) Écrire une fonction `appartient` prenant en argument un tableau d'entiers t et un entier x, et testant si x est dans t.
   (b) En déduire une fonction `mex1` calculant le mex d'un tableau d'entiers naturels.
   (c) Déterminer la complexité, dans le pire cas et le meilleur cas, de la fonction précédente, en fonction de la longueur $n$ du tableau et de son maximum $m$.
3. (a) Écrire une fonction `presences` prenant un tableau t d'entiers naturels et renvoyant un tableau p de $m+1$ booléens, où $m$ est le maximum de t, tel que `p.(k)` indique si k est présent dans t.
   (b) En déduire une autre implémentation `mex2` du minimum exclus.
   (c) Déterminer la complexité dans le pire et le meilleur cas de `mex2`, toujours en fonction de $n$ et $m$.
   (d) Démontrer formellement, à l'aide d'un invariant de boucle, la correction de `mex2` (en supposant que `presences` est correcte).
4. Écrire une variante `mex3` combinant la meilleure complexité dans le meilleur cas et la meilleure complexité dans le pire cas des versions précédentes, en utilisant un dictionnaire. On utilisera les appels suivants (tous de complexité constante) :
   - `hashtbl.create 0` renvoie un dictionnaire vide ;
   - `hashtbl.add h c v` ajoute au dictionnaire `h` la clé `c` associée à la valeur `v` ;
   - `hashtbl.mem h c` teste si la clé `c` apparait dans le dictionnaire `h`.

   On justifiera qu'on obtient bien les complexités désirées.