Exercice 1 : On considère le type de liste natif de OCaml. Écrire des fonctions répondant aux spécifications suivantes. Vous prendrez garde au typage.

1. Une fonction récupérant l'avant dernier élément d'une liste s'il existe.
2. Une fonction récupérant l'indice d'un élément x dans une liste.
3. Une fonction récursive terminale permettant de renverser une liste.
4. Une fonction `maMap` qui applique une fonction `f` à chacun des éléments d'une liste.
5. Une fonction récursive terminale permettant d'accéder au k-ème élément d'une liste s'il existe.

Exercice 2 : Dans cet exercice, on supposera que nos listes sont des listes d'entiers.

1. Rédiger une fonction récursive permettant de récupérer le premier élément d'une liste s'il existe et qui renvoie un message d'erreur "liste vide" sinon.
2. Rédiger une fonction récursive permettant de faire la somme des éléments d'une liste.
3. Rédiger une fonction récursive `dernierElement l` qui retourne le dernier élément de la liste `l` s'il existe.
4. Rédiger une fonction OCaml `ajouterEnQueue l x` permettant d'ajouter l'élément x à la fin de la liste `l`.

Exercice 3 : On considère la fonction OCaml :

```
let rec mystere f l = match l with
  | [] -> []
  | t::q -> (f t)::mystere f q
;;
```

1. Quel est son type ?
2. Déterminer son rôle au moyen d'une phrase.
3. Faire le lien avec une fonction vue en cours.

Exercice 4 : On considère la fonction OCaml :

```
let rec mystere' f = function
  | [] -> []
  | t::q when f t > 1 -> t::(mystere' f q)
  | t::q -> mystere' f q
;;
```

1. Quel est le type de cette fonction ?
2. Quel est son rôle ?
3. En vous inspirant de cette fonction, proposer une fonction renvoyant la liste des éléments d'une liste vérifiant `f(t) = true` où f est une fonction à valeurs booléennes.

Exercice 5 : Proposer une fonction récursive qui prend en entrée deux listes de même type et renvoie une liste qui est la concaténation des deux listes d'entrée (on n'utilisera pas l'opérateur natif de concaténation de OCaml, seulement le filtrage par motif).

Exercice 6 : Écrire une fonction `listIt f l b` de type `('a -> 'b -> 'b)-> 'a list -> 'b-> 'b` qui prend en entrée une fonction f, une liste $l = [a_0, \dots, a_{n-1}]$ et un objet b et qui renvoie la valeur :

$$(f(a_0, f(a_1, f(a_2, \dots f(a_{n-1}, b)))))$$

Exercice 7 : Dans le langage OCaml, il existe une fonction prédéfinie `exists` de type `('a -> bool) -> 'a list -> bool` qui détermine si dans une liste il existe un élément vérifiant une certaine proposition (c'est à dire un élément x tel que $f(x)$ est vraie où f est une fonction à valeurs booléenne).

1. Donner une implémentation possible pour cette fonction en utilisant le filtrage par motif.
2. Implémenter une fonction `pourTout` de type `('a -> bool) -> 'a list -> bool` qui détermine si tous les éléments d'une liste vérifient une certaine propriété.

Exercice 8 : On souhaite écrire une fonction `purge` qui, appliquée à une liste, retourne une liste dans laquelle les doublons ont été éliminés (on rappelle que la fonction prédéfinie `List.mem` détermine si un élément appartient ou pas à une liste).

1. Écrire une première version de purge dans laquelle seule la dernière occurrence de chaque doublon sera conservée. Par exemple, `purge [1; 2; 3; 1; 4; 3; 1]` renverra `[2; 4; 3; 1]`.
2. Écrire une seconde version de purge dans laquelle seule la première occurrence de chaque doublon sera conservée. Cette fois, `purge [1; 2; 3; 1; 4; 3; 1]` renverra `[1; 2; 3; 4]`.

Exercice 9 : Dans cet exercice, on représente les ensembles au moyen de listes. Proposer des fonctions permettant de :

1. Déterminer si un élément x appartient à un ensemble (type `'a-> 'a list -> bool`).
2. Renvoyer l'union des deux ensembles sans tenir compte des doublons.
3. Renvoyer l'intersection des deux ensembles.
4. Renvoyer l'union des deux ensembles en tenant compte des doublons.
5. Analyser la complexité des fonctions proposées.

Exercice 10 : Soit $n \in \mathbb{N}$, proposer des fonctions récursives permettant de calculer l'entier $\lfloor \sqrt{n} \rfloor$:

1. en effectuant un balayage linéaire,
2. en vous inspirant de la recherche dichotomique.

Exercice 11 : Le **tri par insertion** vise à trier dans l'ordre croissant une liste d'entiers. Il repose sur le principe suivant :

- On dispose d'une fonction `insere x l` de type `int -> int list -> int list` qui insère l'élément `x` à sa place dans la liste `l` supposée déjà triée.
- Ensuite on crée une fonction de tri ainsi :
  - si la liste est vide : on ne fait rien,
  - si la liste est de la forme `t::q` on trie récursivement `q` et on insère `t` à sa place dans la liste triée.

1. Implémenter le tri par insertion sur des listes Caml.
2. En ne comptant comme opérations élémentaires que les comparaisons et les opérations `::`, déterminer une relation de récurrence vérifiée dans le pire des cas par le tri par insertion.
3. Résoudre cette relation de récurrence.
4. Si on suppose que la liste est déjà triée, quelle est la complexité du tri par insertion ?

Exercice 12 : Le **tri par sélection** vise à trier dans l'ordre croissant une liste d'entiers. Il repose sur le principe suivant :

- On cherche le minimum de la liste à trier,
- On place ce minimum en tête de liste,
- On trie récursivement la liste privée de son premier élément.

1. Implémenter cette méthode de tri.
2. Si l'on ne compte comme opérations élémentaires que les comparaisons et les opérations `::`, quelle relation de récurrence est vérifiée par la complexité (pire des cas) du tri par sélection ?
3. Résoudre la récurrence.

Exercice 13 : Le **tri à bulles** vise à trier dans l'ordre croissant une liste d'entiers. Il repose sur le principe suivant :

- On parcourt la liste de gauche à droite, en comparant les éléments consécutifs : s'ils ne sont pas dans le bon ordre, on les échange.
- À la fin du parcours :
  - si on a effectué au moins un échange durant ce parcours, on recommence ;
  - sinon, la liste est triée.

1. Implémenter cette méthode de tri.
2. Si l'on ne compte comme opérations élémentaires que les comparaisons et les opérations `::`, quelle relation de récurrence est vérifiée par la complexité (pire des cas) du tri par sélection ?
3. Résoudre la récurrence.