Ce TP est à réaliser intégralement dans le fragment purement fonctionnel de caml.

**Exercice 1.1** (Nombres - échauffement)**.** Écrire en caml des fonctions

1. `fact` telle que `fact n` renvoie la factorielle de l'entier positif `n` ;
2. `nb_bit_pos` telle que `nb_bit_pos n` renvoie le nombre de bits valant 1 dans l'écriture binaire de l'entier positif `n`.

**Exercice 1.2** (Fibonacci)**.** Voici une écriture naïve d'une fonction calculant les termes de la suite de Fibonacci.

```
let rec fibo n =
  (* précondition : n >= 0 *)
  if n <= 1
  then n
  else fibo(n-1) + fibo(n-2)
```

Écrire une nouvelle version de cette fonction ayant une complexité linéaire en le paramètre `n`.
*Indication.* Vous pouvez utiliser une fonction auxiliaire utilisant deux accumulateurs.

**Exercice 1.3** (Chaînes de caractères)**.** Écrire en caml des fonctions

1. `palindrome` telle que `palindrome m` renvoie `true` si et seulement si la chaîne de caractères `m` est un palindrome (c'est-à-dire qu'on voit la même séquence de caractères en la lisant de gauche à droite ou de droite à gauche) ;
2. `compare` telle que `compare m1 m2` renvoie `true` si et seulement si la chaîne de caractères `m1` est plus petite dans l'ordre lexicographique que la chaîne de caractères `m2` (c'est-à-dire que `m1` apparaîtrait avant `m2` dans un dictionnaire) ;
3. `facteur` telle que `facteur m1 m2` renvoie `true` si et seulement si la chaîne de caractères `m1` est un facteur de la chaîne de caractères `m2` (c'est-à-dire que `m1` apparaît telle quelle dans `m2`).

**Exercice 1.4** (Tri fusion)**.** L'algorithme de tri fusion trie une liste en appliquant le principe suivant :

a. couper la liste en deux parties à peu près égales ;
b. trier récursivement chacune des deux listes obtenues ;
c. fusionner les deux listes triées en préservant l'ordre.

Écrire des fonctions

1. `split` telle que `split l` renvoie deux listes obtenues en partageant les éléments de la liste `l` de manière aussi équilibrée que possible ;
2. `merge` telle que `merge l1 l2` renvoie une liste contenant les éléments des listes `l1` et `l2` triés par ordre croissant, en supposant que chacune des listes `l1` et `l2` passée en paramètre est elle-même triée par ordre croissant ;
3. `tri` telle que `tri l` renvoie une liste contenant les éléments de la liste `l` triés par ordre croissant.

**Exercice 1.5** (Listes)**.** Écrire des fonctions

1. `somme_carres` telle que `somme_carres l` renvoie la somme des carrés des éléments de la liste d'entiers `l` ;
2. `find_opt` telle que `find_opt x l` renvoie `Some i` si l'élément `x` apparaît à l'indice `i` de la liste `l` (mais pas avant), et `None` si `x` n'apparaît pas dans `l`.

Refaire l'exo sans utiliser le mot-clé `rec`. À la place, usez et abusez des fonctions de la bibliothèque `List` de caml.

**Exercice 1.6** (Récursion terminale)**.** Créer une liste `l` contenant dans l'ordre les entiers positifs de zéro à un million.

Écrire ensuite des fonctions `rev` et `map` correspondant aux fonctions `List.rev` et `List.map` de caml. Vous devrez faire en sorte que ces fonctions puissent s'appliquer à la liste `l` précédente sans provoquer de débordement de pile.

*Bonus.* Réécrivez les fonctions des exercices précédents pour les rendre récursives terminales, si pertinent.

**Exercice 1.7** (Concaténation)**.** Voici une manière de coder la concaténation de deux listes en caml.

```
let rec concat l1 l2 = match l1 with
  | [] -> l2
  | x::s -> x :: (concat s l2)
```

Cette fonction, comme l'opérateur `@` fourni par caml, a un coût proportionnel à la longueur de la première liste. Pour pouvoir effectuer de multiples concaténations sans crainte de leur coût, on propose une nouvelle représentation des séquences, basée sur le type de données suivant (qu'on peut voir comme un arbre de concaténations).

```
type 'a seq =
  | Elt of 'a
  | Seq of 'a seq * 'a seq
```

La concaténation de deux séquences `s1` et `s2` est donc simplement `Seq(s1, s2)`. On peut se donner un raccourci d'écriture `s1 @@ s2` avec la définition

```
let (@@) x y = Seq(x, y)
```

Un tel arbre représente une séquence, obtenue en considérant tous ses éléments dans l'ordre de gauche à droite. Les deux arbres `Seq(Elt 1, Seq(Elt 2, Elt 3))` et `Seq(Seq(Elt 1, Elt 2), Elt 3)` sont les deux représentations possibles de la liste `[1; 2; 3]`.

Écrire pour cette structture de séquence les fonctions suivantes :

1. `hd`, `tl`, `mem`, `rev`, `map`, `fold_left`, `fold_right` correspondant aux fonctions de même nom sur les listes ;
2. `seq2list` telle que `seq2list s` renvoie une liste caml représentant la séquence `s` (ne pas utiliser `@`) ;
   *Bonus (difficile) :* donner une version récursive terminale de cette fonction ;
3. `find_opt` telle que `find_opt x s` renvoie `Some i` si l'élément `x` apparaît à la position `i` dans la séquence représentée par `i` (mais pas avant) et `None` si `x` n'apparaît pas dans `s` ;
4. `nth` telle que `nth s n` renvoie l'élément d'indice `n` dans `s` (et lève une exception si l'indice ne convient pas).

Comment enrichir notre structure de séquence pour rendre potentiellement plus efficace la fonction `nth` ? Définir le nouveau type correspondant et redéfinir les fonctions `(@@)` et `nth` en conséquence. D'autres fonctions doivent-elles encore être mises à jour ?