Exercice 1 :

On considère la fonction suivante :

```
1 let rec mystere p l =
2   match l with
      | [] -> false
4     | x::q -> p x || mystere p q
```

1. On voit que `l` est une `'a list`, compte tenu du filtrage. Donc à la ligne 4, `x` est de type `'a`.
   Comme `p` est appliqué à `x` et qu'on utilise l'opérateur `||` sur le résultat, on a `p : 'a -> bool`.
   Enfin, la ligne 3 et la ligne 4 indiquent toutes les deux que le résultat de la fonction est de type `bool`.
   On a donc `mystere : ('a -> bool) -> 'a list -> bool`.
2. (a) La fonction `mystere` prend en argument p une propriété sur un certain type, et $\ell = [x_0; \dots; x_{n-1}]$ une liste d'éléments de ce type, et indique si la formule $\exists i \in [\![0; n-1]\!], p(x_i)$ est vraie.
   (b) Par exemple, si on pose `let p x = x > 0`, on a :
   - `mystere p [1; 2; -3; 3; 1] = false`
   - `mystere p [1; 3; 3; 1` = true|
   (c) Montrons la correction par récurrence sur $n \geqslant 0$ la taille de la liste l.
   - *Init :* Si $n = 0$, la liste est vide donc il ne peut exister aucun élément qui vérifie p. La fonction est correcte.
   - *Hérédité :* Soit $n \geqslant 1$ tel que pour toute liste de taille $n-1$, la fonction est correcte. Soit ℓ de taille n.
     ℓ est non vide, on pose donc $\ell = x::q$.
     Si p vérifie x, alors il existe un élément de ℓ que p vérifie, et donc il est correct que la fonction renvoie `true`.
     Sinon, alors p vérifie un élément de ℓ si et seulement si p vérifie un élément de q. Or on sait par hypothèse de récurrence que `mystere p q` est correcte, c'est à dire faut `true` si et seulement si p vérifie un élément de q. Donc il est correct que la fonction renvoie `mystere p q`.

Exercice 2 :

```
1 let rec ajout l x =
2   match l with
      | [] -> [x]
4     | y::q ->
          if y = x then l
6         else if x < y then x::l
          else (* y < x *) y::(ajout q x)

1 let rec retire l x =
2   match l with
      | [] -> []
4     | y::q ->
          if y >= x then q
6         else (* y < x *) y::(retire q x)
```

3.
```
let rec intersection l1 l2 =
  match (l1, l2) with
    | ([], _) -> []
    | (_, []) -> []
    | (x1::q1, x2::q2) ->
         if x1 = x2 then x1::(intersection q1 q2)
         else if x1 < x2 then intersection q1 l2
         else (* x2 < x1 *) intersection l1 q2
```

4. On donne la fonction suivante

```
let rec union l1 l2 =
  match (l1, l2) with
    | ([], _)          -> l2
    | (_, [])          -> l1
    | (x1::q1, x2::q2) ->
        if x1 = x2 then x1::(union q1 q2)
        else if x1 < x2 then x1::(union q1 l2)
        else (* on a x2 < x1 *) x2::(union l1 q2)
```

(a) Montrons par récurrence sur $m \geqslant 0$ que étant donné deux listes ℓ_1 et ℓ_2 de tailles m_1 et m_2 avec $m_1 + m_2 = m$, `union` est correcte sur ℓ_1 et ℓ_2.

- *Init :* Si $m = 0$, $m_1 = m_2 = 0$ et la fonction renvoie la liste vide, ce qui est correct.
- *Hérédité :* Soit $m \geqslant 1$. Supposons la fonction correcte sur toutes listes de taille totale $m' \in [\![0; m-1]\!]$.

  Soit ℓ_1 et ℓ_2 des listes de taille m_1 et m_2 telles que $m_1 + m_2 = m$.

  Si $m_1 = 0$ ou $m_2 = 0$, l'une des listes est vide et il est correct de simplement renvoyer l'autre.

  On suppose désormais les deux listes non vides : on a $\ell_1 = x_1 :: q_1$ et $\ell_2 = x_2 :: q_2$.

  Si $x_1 = x_2$, alors l'union est composée du singleton $\{x_1\}$, qui est le minimum, et de l'union des ensembles représentés par q_1 et q_2. Or la taille totale de q_1 et q_2 est $m_1 - 1 + m_2 - 1 = m - 2 \in [\![0; m-1]\!]$. Donc par hypothèse de récurrence, le résultat est correct.

  Sinon, si $x_1 < x_2$, alors le minimum de l'union est x_1, et les autres éléments sont l'union de l'ensemble représenté par q_1 et de l'ansemble représenté par ℓ_2 (aucun des deux ensembles ne comprend x_1. Or la taille totale de q_1 et ℓ_2 est $m_1 - 1 + m_2 = m - 1$. Encore une fois, par hypothèse de récurrence, le résultat est correct.

  Le cas $x_2 < x_1$ est simplement le symétrique du précédent.

(b) Dans le pire des cas, on est toujours dans l'un des deux derniers cas, i.e. on fait un appel récursif sur deux listes dont la taille totale n'a diminué que de 1. Ceci se produit par exemple si $\ell_1 = [1; 3; \ldots; 2k-1]$ et $\ell_2 = [0; 2; \ldots; 2k-2]$.

Comme chaque appel coûte un nombre au plus constant d'opérations atomique, en plus du coût de l'appel récursif, on a un coût total proportionnel au nombre d'appels effectués, c'est à dire $\mathcal{O}(m_1 + m_2)$.

5. (a) On définit (en fonction de n) une fonction `etend` qui, étant donné une liste de parties de F_{n-1}, calcule la liste des parties de F_n obtenues en ajoutant n à chacun de ces ensembles.

```
let rec parties n p =
  if n < p  then []
  else if p = 0 then [[]]
  else
    let rec etend l =
      match l with
        | [] -> []
        | a::q -> (a @ [n])::(etend q)
    in
    (parties (n - 1) p) @ (etend (parties (n - 1) (p - 1)))
```

(b) `parties 3 2` :

- commence par calculer `parties 2 2` qui est la concaténation de :
  - `parties 1 2`, qui est la liste vide
  - `etend (parties 1 1)` avec $n = 2$. Or `parties 1 1` est la concaténation de :
    - `parties 0 1 = []`
    - `etend (parties 0 0)` avec $n = 1$, c'est à dire, `etend [[]]`, donc `[1]`.

    Donc `etend (parties 1 1) = [[1; 2]]`

  Donc `parties 2 2 = [] @ [[1; 2]] = [[1; 2]]`
- calcule ensuite `etend (parties 2 1)` avec $n = 3$. Or `parties 2 1` est la concaténation de :
  - `parties 1 1 = [[1]]` (même calcul que plus haut)
  - `etend (parties 1 0)` avec $n = 2$, c'est à dire `etend [[]] = [[2]]`.

  Donc `etend (parties 2 1) = [[1; 3]; [2; 3]]`

Il en résulte qu'on a bien le résultat final `[[1; 2]; [1; 3]; [2; 3]]`.

Exercice 3 :

```
let ajoute t x = t.(x) <- true

let retire t x = t.(x) <- false

let union t1 t2 =
  let n = Array.length t1 in (* et aussi longueur de t2 par hypothèse *)
  for i = 0 to n - 1 do
    t1.(i) <- t1.(i)  || t2.(i)
  done

let union_copie t1 t2 =
  let n = Array.length t1 in (* et aussi longueur de t2 par hypothèse *)
  let t3 = Array.make n false in
  for i = 0 to n - 1 do
    t3.(i) <- t1.(i)  || t2.(i)
  done;
  t3
```