2 Fonctions de base sur les listes

1.
```
let rec length = function
    | [] -> 0
    | x :: q -> 1 + length q;;

let rec mem x = function
    | [] -> false
    | y :: q -> (x = y) || (mem x q);;

let rec append l1 l2 = match l1, l2 with
    | [], _ -> l2
    | x :: q, _ -> x :: (append q l2);;

let rec map f = function
    | [] -> []
    | x :: q -> (f x) :: (map f q);;

let rec iter f = function
    | [] -> ()
    | x :: q -> f x; iter f q;;
```

2.
```
let rec last = function
    | [] -> failwith "liste vide"
    | [x] -> x
    | x :: q -> last q;;
```

3.
```
let rec nth l n = match l, n with
    | [], _ -> failwith "Liste trop courte"
    | x :: q, 0 -> x
    | x :: q, _ -> nth q (n - 1);;
```

4.
```
let rec sum = function
    | [] -> 0
    | x :: q -> x + sum q;;
```

5.
```
let rec sorted = function
    | [] -> true
    | [x] -> true
    | x :: y :: q -> (x <= y) && sorted (y :: q);;
```

6.
```
let max l =
   let rec aux_max m = function
      | [] -> m
      | x :: q when x <= m -> aux_max m q
      | x :: q -> aux_max x q in
   match l with
      | [] -> failwith "liste vide"
      | x :: q -> aux_max x q;;
```

7.
```
let rev l =
   let rec aux_rev acc = function
      | [] -> l
      | x :: q -> aux_rev (x :: acc) q in
   aux_rev [] l;;
```

8.
```
let flatten ll =
   let rec aux_flatten acc = function
      | [] -> acc
      | [] :: ql -> aux_flatten acc ql
      | (x :: q) :: ql -> x :: (aux_flatten acc (q :: ql)) in
   aux_flatten [] ll;;
```

9.
```
let rec partition p = function
   | [] -> [], []
   | x :: q -> let l1, l2 = partition p q in
               if p x then x :: l1, l2 else l1, x :: l2;;
```

10.
```
let readint l =
   (* On commence par transformer la liste en
   une liste de chiffres *)
   let rec digit_list = function
      | [] -> []
      | x :: q when x < 10 -> x :: (digit_list q)
      | x :: q -> digit_list (x / 10 :: x mod 10 :: q) in
   (* Ensuite, on la concatène en un seul nombre *)
   let rec aux_readint = function
      | [] -> 0, 1
      | x :: q -> let n, m = aux_readint q in
                  m * x + n, m * 10 in
    fst (aux_readint (digit_list l));;
```

11.
```
let rec sublist l1 l2 = match l1, l2 with
   | [], _ -> true
   | x :: q, [] -> false
   | x :: q, y :: r when x = y -> sublist q r
   | _, y :: r -> sublist l1 r;;
```

12.
```
let rec pattern l1 l2 =
   (* On utilise une fonction auxiliaire qui
   teste si l1 est préfixe de l2 *)
   let rec prefixe l1 l2 = match l1, l2 with
      | [], _ -> true
      | x :: q, y :: r when x = y -> prefixe q r
      | _ -> false in
   match l1, l2 with
      | [], _ -> true
      | x :: q, [] -> false
      | x :: q, y :: r when x = y -> (prefixe l1 l2) || (pattern l1 r)
      | x :: q, y :: r -> pattern l1 r;;
```

3 Des suites et des listes

1.
```
let rec geo q u0 n = match n with
   | 0 -> [u0]
   | n -> u0 :: (geo q (q * u0) (n - 1));;
```

2.
```
let rec rec_seq f u0 = function
   | 0 -> [u0]
   | n -> u0 :: (rec_seq f (f u0) (n - 1));;
```

3.
```
let fibo n =
   (* On utilise une fonction auxiliaire qui calcule
   les termes deux par deux *)
   let rec fibaux u0 u1 = function
      | 0 -> [u0; u1]
      | n -> u0 :: (fibaux u1 (u0 + u1) (n - 1)) in
   match n with
      | 0 -> [0]
      | n -> fibaux 0 1 (n - 1);;
```

4. Dans un premier temps, remarquons qu'un terme de la suite ne contient que des valeurs inférieures ou égales à 3, et donc jamais plus de 3 chiffres consécutifs identiques (cela se prouve facilement).
On crée une fonction permettant de « lire » un terme de la suite pour trouver le suivant, et on fait un appel à la fonction `rec_seq`.

```
let rec f n =
   if n = 0 then 0 else begin
   let a = n mod 10 in
   let b = (n mod 100) / 10 in
   let c = (n mod 1000) / 100 in
   match (a = b, b = c) with
      | false, _ -> 10 + a + (f (n / 10)) * 100
      | true, false -> 20 + a + (f (n / 100)) * 100
      | _ -> 30 + a + (f (n / 1000)) * 100
   end;;

let look_say = rec_seq f 1;;
```

5. Pour $n = 10$, on trouve le résultat suivant :

```
# look_say 10;;
- : int list =
[1; 11; 21; 1211; 111221; 312211; 13112221; 1113213211; 31131211131221;
 3987939086258336403; 4363847194894301653]
```

Le problème qui se pose ici est que les calculs se font modulo 2^{62}, ce qui pose souci pour des très grands termes.

6.
```
let rec g = function
    | [] -> []
    | [x] -> [1; x]
    | [x; y] when x = y -> [2; x]
    | [x; y] -> [1; x; 1; y]
    | x :: y :: q when x <> y -> 1 :: x :: (g (y :: q))
    | x :: y :: z :: q when y <> z -> 2 :: x :: (g (z :: q))
    | x :: y :: z :: q -> 3 :: x :: (g q);;

let look_say = rec_seq g [1];;
```