1) On se rappelle que la fonction `mod : int -> int -> int` permet de calculer le modulo :
```
let f u = (123 * u + 15) mod 32767;;
```

2) C'est une simple fonction récursive, en suivant la définition :

```
let rec alea n u0 =
  if n = 0 then []
  else u0 :: (alea (n-1) (f u0))
;;

let l = alea 10 1594;;
val l : int list =
  [1594; 32242; 974; 21516; 25123; 10046; 23294; 14448; 7701; 29762]
```

3)
```
let rec map f = function
  | [] -> []
  | t :: q -> (f t) :: (map f q)
;;

map (fun x -> x mod 3) l;;
- : int list = [1; 1; 2; 0; 1; 2; 2; 0; 0; 2]
```

4) Si on utilise `@` il n'y a rien à faire, donc on le fait manuellement. Cette fonction a une complexité linéaire dans la taille de la liste `l1`.

```
let rec conc l1 l2 = match l1 with
  | [] -> l2
  | t :: q -> t :: (conc q l2)
;;
```

5)
```
let rev1 l =
  let rec retourne accu = function
    | [] -> accu
    | t::q -> retourne (t::accu) q
  in retourne [] l
;;
```
Notez que l'on doit dire “renvoyer” une valeur, et pas “retourner” une valeur, puisque “retourner” signifie inverser une liste, comme pour cette question.

6)
```
let rec filtre p = function
  | [] -> []
  | t :: q -> let fq = filtre p q in
              if p t then t :: fq else fq
;;

let p x = x mod 2 = 1 in (* un exemple de predicat *)
  filtre p l;;
```

7)
```
let rec fibogen gamma f0 f1 n =
```

```
  if n=0 then f0
  else if n=1 then f1
  else fibogen gamma f1 (gamma f0 f1) (n-1)
;;
```

8)
```
(+);; (* notation infixe *)
- : int -> int -> int = <fun>

let fibo_usuel = fibogen (+) 0 1;;
- : val fibo_usuel : int -> int = <fun>

fibo_usuel 12;; (* On retrouve F_12 = 144 *)
- : int = 144
```

9) La concaténation de chaînes de caractères se fait avec ^ (accent circonflexe) :

```
let fiboword =
  let gamma s1 s2 = s2 ^ s1
  in fibogen gamma "b" "a"
;;
- : val fiboword : int -> string = <fun>

fiboword 5;;
- : string = "abaababa"
```

10)
```
let fibolist =
  let gamma l1 l2 = l2 @ l1 in
  fibogen gamma [1] [0];;
- : val fibolist : int -> int list = <fun>

fibolist 5;;
- : int list = [0;1;0;0;1;0;1;0]
```

11)
```
let rec nb_occ x = function
  | [] -> 0
  | t::q -> (if t=x then 1 else 0) + nb_occ x q
;;
```

Notons k_n le nombre d'occurrences de 0 dans l_n, et notons F_n le terme d'indice n de la suite de Fibonacci usuelle. Il est clair que $k_0 = F_0(= 0)$ et que $k_1 = F_1(= 1)$. Soit $n \geqslant 2$, supposons que $F_{n-1} = k_{n-1}$ et $F_{n-2} = k_{n-2}$. Alors $l_n = l_{n-1}@l_{n-2}$ donc $k_n = k_{n-1} + k_{n-2} = F_{n-1} + F_{n-2} = F_n$. Par récurrence (double) pour tout $n \in \mathbb{N}, F_n = k_n$.

12) a) On utilise une fonction auxiliaire récursive `ne_garder_que : int -> 'a list -> 'a list` telle que l'appel `ne_garder_que i l` renvoie une liste que ne contient que les `i` premiers éléments de `l` (ceci permet de ne calculer la longueur de la liste qu'une seule fois et de garantir une complexité O(longueur)).

```
let tronque n l =
  let rec ne_garder_que i l = match l with
    | [] -> []
```

```
      | _ when i <= 0 -> []
      | t :: q -> t :: (ne_garder_que (i-1) q)
    in ne_garder_que (List.length l - n) l
  ;;
```

b) Puisque pour tout $n \geqslant 2, l_n$ termine comme l_{n-2}, puisque l_2 termine par [0;1] et l_3 termine par [1;0], il vient pour tout $n \geqslant 2$: l_n termine par $\begin{cases}[0;1] & \text{si } n \text{ pair,}\\ [1;0] & \text{si } n \text{ impair.}\end{cases}$

Notons donc pour tout $n \geqslant 2$: $f_n = \begin{cases}[0;1] & \text{si } n \text{ pair,}\\ [1;0] & \text{si } n \text{ impair.}\end{cases}$

Notons aussi l'_n la liste l_n privée de ses deux derniers éléments ; ainsi $l_n = l'_n @ f_n$.

$l'_2 = [\,]$, $l'_3 = [0]$ et $l'_4 = [0;1;0]$ sont des palindromes. Soit $n \geqslant 5$, supposons que l'_{n-1}, l'_{n-2} et l'_{n-3} sont des palindromes. Alors :

$$l'_n = l_{n-1} @ l'_{n-2} = l_{n-2} @ l_{n-3} @ l'_{n-2} = l'_{n-2} @ f_{n-2} @ l'_{n-3} @ f_{n-3} @ l'_{n-2}$$

Or l'_{n-2} et l'_{n-3} sont des palindromes, et f_{n-2} est le miroir de f_{n-3} donc l'_n est un palindrome.

13) a)
```
let rec remplace x lx = function
  | [] -> []
  | t :: q -> if x=t then lx @ (remplace x lx q)
              else t :: (remplace x lx q)
;;
```

b) Notons σ la substitution où on remplace tous les `0` par `[0;1]` et `1` par `[0]`.

Pour effectuer σ avec la fonction `remplace` on peut par exemple commencer par remplacer les 1 par des [2], puis les 0 par des [0;1], enfin les 2 par des 0.

Prouvons par récurrence double que $\forall n \in \mathbb{N}, \sigma(l_n) = l_{n+1}$.

$\sigma(l_0) = \sigma([1]) = [0] = l_1$; et $\sigma(l_1) = \sigma([0]) = [0;1] = l_2$.

Soit $n \geqslant 2$, supposons que $\sigma(l_{n-2}) = l_{n-1}$ et que $\sigma(l_{n-1}) = l_n$. Alors $\sigma(l_n) = \sigma(l_{n-1} @ l_{n-2}) = \sigma(l_{n-1}) @ \sigma(l_{n-2}) = l_n @ l_{n-1} = l_{n+1}$.□

14)
```
let successeur l = ()::l;;
```

15)
```
let rec entiers_vers_batons = function
  | 0 -> []
  | n -> successeur (entiers_vers_batons (n-1))
;;

let rec batons_vers_entiers = function
  | [] -> 0
  | () :: q -> 1 + (batons_vers_entiers q)
;;
```

16) a)
```
let rec inferieur l1 l2 = match l1, l2 with
  | _, [] -> false (* _ est une variable sans nom *)
  | [], _ -> true
  | ()::q1, ()::q2 -> inferieur q1 q2
;;
```

b) `let rec somme l1 l2 = l1@l2;;`

c)
```
let rec difference l1 l2 = match (l1,l2) with
  | [], _ -> []
  | _, [] -> l1
  | _::q1, _::q2 -> difference q1 q2
;;
```

d) On traduit la formule mathématique $\forall n_1, n_2 \in \mathbb{N}, n_1 n_2 = \begin{cases} 0 & \text{si } n_2 = 0 \\ n_1 + n_1(n_2 - 1) & \text{sinon} \end{cases}$

```
let rec produit l1 = function
  | [] -> []
  | _ :: q -> somme l1 (produit l1 q)
;;
```

e) Soient $n_1, n_2 \in \mathbb{N}$; on veut effectuer la division euclidienne de n_1 par n_2.
Si $n_1 < n_2$ alors le quotient est nul et le reste est n_1 car $n_1 = 0.n_2 + n_1$
Sinon, soient q le quotient et r le reste de la division euclidienne de $n_1 - n_2$ par n_2. Alors $r < n_2$ et $n_1 = (q+1)n_2 + r$.

```
let rec division l1 l2 =
  if inferieur l1 l2 then ([], l1)
  else let q, r = division (difference l1 l2) l2 in
        successeur q, r
;;
```

f) On applique l'algorithme (usuel) de décomposition en base b, poids faible en tête :

```
let rec baseb l b =
  if l=[] then []
  else let q, r = division l b in
        r::(baseb q b)
;;
```

17) a) Son cas de base est quand le dernier paramètre est la liste vide : on se contente alors de renvoyer la valeur `a`, sans appliquer la fonction `f`.

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

18) a) On utilise encore la fonction `(+)` : `let somme_liste = fold_left (+) 0;;`

b)
```
let produit_liste =
  (* on peut aussi utiliser : let f = ( * ) notation infixe *)
  let f a b = a * b in
  fold_left f 1
;;
```

c) `let max_liste = fold_left max min_int;;`

19) `let flatten = fold_left (@) [];;`

20)
```
let reverse =
  let f l x = x :: l in (* aussi : ( :: ) *)
  fold_left f []
;;
```

21)
```
let rec fold_left2 g a lx ly = match lx, ly with
  | [], [] -> a
  | x :: qx, y :: qy -> fold_left2 g (g a x y) qx qy
  | _ -> failwith "Listes de tailles differentes"
;;

let palindrome l =
  let g b x y = b && (x = y) in
  fold_left2 f true g (reverse l)
;;
```

22) L'idée est de procéder par "force brute" : on teste toutes les listes d'entiers extraites de $[\![1, 20]\!]$ (il y en a environ un million).

Il y a 36 solutions dont par exemple $[4; 6; 9; 13; 14; 15; 17; 20]$.