

DEVOIR SURVEILLÉ N° 1
Éléments de correction
EXERCICE 1

1.

```
let rec liste_decroissante n =
  match n with
  | 0 -> []
  | _ -> n::liste_decroissante (n-1)
;;
```
2.

```
let liste_croissante n =
  let rec aux k acc =
    match k with
    | 0 -> acc
    | _ -> aux (k-1) (k::acc)
  in aux n []
;;
```

EXERCICE 2

1.

```
let p = {nom = "ALCAZAR" ;
  prenom = "Peggy" ;
  classe = MPSI 3 ;
  sexe = F} ;;
```

2. Deux propositions :

<pre>let deuxieme_annee e = match e.classe with MPetoile -> true MP _ -> true _ -> false ;;</pre>	<pre>let deuxieme_annee e = match e.classe with MPSI _ -> false _ -> true ;;</pre>
--	--

3. Une première solution où on récupère le résultat de l'appel récursif :

```
let rec etude_parite lst =
  match lst with
  | [] -> (0, 0)
  | t::q -> let a, b = etude_parite q in
```

```
  match t.sexe with
  | F -> a+1, b
  | M -> a, b+1
```

```
;;
```

Une autre solution où on définit une fonction auxiliaire avec deux accumulateurs :

```
let etude_parite lst =
  let rec aux lst f m =
    match lst with
    | [] -> (f, m)
    | t::q -> if t.sexe = F
      then aux q (f+1) m
      else aux q f (m+1)
  in aux lst 0 0
;;
```

EXERCICE 3

1. (a) 0 est pair. Soit $a = \sum_{i=0}^n 2^i a_i$, alors $a = a_0 + 2 \left(\sum_{i=1}^n 2^{i-1} a_i \right)$: a est pair si et seulement si $a_0 = 0$.

```
let estPair a =
  match a with
  | [] -> true
  | 0::q -> true
  | _ -> false
;;
```

- (b)

```
let egalUn a =
  a = [1]
;;
```

- (c) Si $a = 0$, $2a = 0$. Si $a = \sum_{i=0}^n 2^i a_i$, alors $2a = \sum_{i=0}^n 2^{i+1} a_i = 0.2^0 + \sum_{i=1}^{n+1} 2^i a_{i-1}$.

```
let double a =
  match a with
  | [] -> []
  | _ -> 0::a
;;
```

(d) Si $a = 0$, $\lfloor a/2 \rfloor = 0$. Si $a = \sum_{i=0}^n 2^i a_i$, alors $\lfloor a/2 \rfloor = \sum_{i=1}^n 2^{i-1} a_i = \sum_{i=0}^{n-1} 2^i a_{i+1}$.

```
let moitie a =
  match a with
  | [] -> []
  | t::q -> q
;;
```

2. (a) Si $a = 0$, alors $a + 1 = 1$.

Si $a = \sum_{i=0}^n 2^i a_i$ avec $a_0 = 0$, alors $a + 1 = 1 + \sum_{i=1}^n 2^i a_i$.

Si $a = \sum_{i=0}^n 2^i a_i$ avec $a_0 = 1$, alors $a + 1 = 2 + \sum_{i=1}^n 2^i a_i = 2 \times \left(1 + \sum_{i=1}^n 2^{i-1} a_i\right)$

(il y a une retenue).

```
let rec successeur a =
  match a with
  | [] -> [1]
  | 0::q -> 1::q
  | _::q -> 0::(successeur q)
;;
```

(b) Si $a = 0$, alors $a + b = b$ et si $b = 0$, alors $a + b = a$.

Si $a = \sum_{i=0}^n 2^i a_i$ et $b = \sum_{i=0}^n 2^i b_i$, alors :

- ou bien $a_0 = 0$ (resp. $b_0 = 0$) et $a + b = b_0 + \left(\sum_{i=1}^n 2^i a_i + \sum_{i=1}^n 2^i b_i\right)$ (resp.

$$a + b = a_0 + \left(\sum_{i=1}^n 2^i a_i + \sum_{i=1}^n 2^i b_i\right).$$

- ou bien $a_0 = b_0 = 1$

$$\text{et } a + b = 2 + \left(\sum_{i=1}^n 2^i a_i + \sum_{i=1}^n 2^i b_i\right) = 2 \times \left(1 + \sum_{i=1}^n 2^{i-1} a_i + \sum_{i=1}^n 2^{i-1} b_i\right).$$

```
let rec add a b =
  match a, b with
  | [], _ -> b
  | _, [] -> a
```

```
| 0::qa, b0::qb -> b0::(add qa qb)
| a0::qa, 0::qb -> a0::(add qa qb)
| _::qa, _::qb -> 0::(successeur (add qa qb))
;;
```

3. (a) `let rec mult a b =`

```
  match b with
  | [] -> []
  | [1] -> a
  | _ -> if estPair b
  then mult (double a) (moitie b)
  else add a (mult (double a) (moitie b))
;;
```

(b) La fonction `mult` termine lorsque la liste `b` est de longueur 0 ou 1.

Supposons que la fonction `mult` termine pour toute liste `b` de longueur $n \in \mathbb{N}^*$. Alors si `b` est une liste de longueur $n + 1$, `moitie b` renvoie une liste de longueur n donc l'appel `mult (double a) (moitie b)` termine ; par conséquent, l'appel `mult a b` termine (sous réserve de terminaison de `add...`).

Finalement, la fonction `mult` termine.

(c) Lorsque la liste `b` est de longueur 0, alors elle représente 0, et `mult a b` renvoie `[]` qui représente 0, ce qui est correct.

Lorsque la liste `b` est de longueur 1, alors elle est égale à `[1]` et représente 1, et `mult a b` renvoie `a`, ce qui est correct.

Supposons que `mult a b` renvoie bien la représentation de ab pour toute liste `b` de longueur $n \in \mathbb{N}^*$. Alors si `b` est une liste de longueur $n + 1$, `moitie b` renvoie une liste de longueur n donc `mult (double a) (moitie b)` renvoie la représentation de $2a \times \left\lfloor \frac{b}{2} \right\rfloor$ et

- si b est pair, $2a \left\lfloor \frac{b}{2} \right\rfloor = ab$ et `mult a b` renvoie bien la représentation de ab ;
- si b est impair, $2a \left\lfloor \frac{b}{2} \right\rfloor = a(b-1)$ auquel on ajoute a , donc `mult a b` renvoie bien la représentation de ab .

Finalement, la fonction `mult` calcule bien le produit ab .

(d) Notons C_n le nombre total d'appels en fonction de la longueur de la liste `b`, alors $C_0 = C_1 = 1$ et pour tout $n \geq 2$, $C_n = 1 + C_{n-1}$, donc la suite $(C_n)_{n \in \mathbb{N}}$

est arithmétique à partir du rang 1 et pour tout $n \in \mathbb{N}^*$, $C(n) = n$.

Comme n est le nombre de bits de `b`, la complexité de `mult` est donc en $O(\log b)$.

- (e) La fonction `mult` n'est pas récursive terminale, car dans le cas où b est impair, on appelle la fonction `add` avec pour argument le résultat de l'appel récursif de `mult` : l'appel récursif n'est alors pas la dernière opération effectuée.

EXERCICE 4

1. (a)

```
let rec ieme i lst =
  match lst with
  | [] -> failwith "Longueur_insuffisante"
  | t::q -> if i = 1
    then t
    else ieme (i-1) q
;;
```

(b)

```
let rec tete i lst =
  match lst, i with
  | [], _ -> []
  | _, 0 -> []
  | t::_, 1-> [t]
  | t::q, _ -> let tq = tete (i-1) q in
    if tq = [] then [] else t::tq
;;
```

(c)

```
let rec queue i lst =
  match lst, i with
  | [], _ -> []
  | _, 0 -> lst
  | t::q, _ -> queue (i-1) q
;;
```

(d)

```
let rec ajoute x lst =
  match lst with
  | [] -> [x]
  | t::q -> t::(ajoute x q)
;;
```

2.

```
let rec calcule_j lst =
  match lst with
  | [] -> 0
  | [_] -> 1
  | t1::q -> let j2 = calcule_j q in
```

```
  if t1 > List.hd q + 2
  then j2 + 1
  else
    if j2 = 1
    then 1
    else j2 + 1
;;
```

```
let suivant (j, lst) =
  let lgr = List.length lst in
  let kr = ieme lgr lst in
  if kr > 3
  then let lst_suivant = ajoute 2 lst in
    let new_j = (if kr = 4 then j else List.length lst_suivant)
    in (new_j, lst_suivant)
  else
    if j = 1
    then (1, [List.hd lst + 1])
    else
      let lst_suivant = ajoute (ieme j lst + 1) (tete (j-1) lst) in
      (calcule_j lst_suivant, lst_suivant)
;;
```

```
let enumere n =
  let rec aux last acc nb =
    match nb with
    | 0 -> acc
    | _ -> let next = suivant last in
      aux next (ajoute next acc) (nb-1)
  in
  let un = (1, [2]) in
  aux un [un] (n-1)
;;
```

3.

```
let rec insere e lst =
  match lst with
  | [] -> [e]
  | t::q -> if t > e
    then t::(insere e q)
    else e::lst
```

```

;;

let rec une_etape lst =
  match lst with
  | [] -> [], true
  | [_] -> lst, true
  | t1::t2::q ->
    match t1-t2 with
    | 0 ->
      begin
        match t1 with
        | 2 -> 3::q, false
        | 3 -> 4::(insere 2 q), false
        | _ -> (t1+1)::(insere (t1-2) q), false
      end
    | 1 -> (t1+1)::q, false
    | _ -> let qr, fini = une_etape (t2::q) in
      t1::qr, fini
;;

let rec decomposition liste =
  let lst, fini = une_etape liste in
  if fini
  then lst
  else decomposition lst
;;

4. let somme a b =
  let rec fusion l1 l2 =
    match l1,l2 with
    | [],_ -> l2
    | _,[] -> l1
    | t1::q1,t2::q2 when t1 > t2 -> t1::(fusion q1 l2)
    | _ ,t2::q2 -> t2::(fusion l1 q2)
  in
  decomposition (fusion a b)
;;

```