

**Exercice 1 (listes) :**

1. 

```
let rec longueur l = match l with
| [] -> 0
| _::q -> 1 + longueur q
```
2. 

```
let rec compter x l = match l with
| [] -> 0
| t::q when t = x -> 1 + compter x q
| _::q -> compter x q
```
3. 

```
let rec iter f l = match l with
| [] -> ()
| t::q -> f t ; iter f q
```
4. 

```
let rec filter f l = match l with
| [] -> []
| t::q when f t -> t :: filter f q
| _::q -> filter f q
```
5. 

```
let rec append l1 l2 = match l1 with
| [] -> l2
| t::q -> t :: append q l2
```
6. 

```
let rec flatten l = match l with
| [] -> []
| t::q -> append t (flatten q)
```

**Exercice 2 (tableaux et exceptions) :**

1. 

```
let somme_produit a b =
  let n = Array.length a in
  if Array.length b <> n then raise Mauvaises_longueurs ;
  let s = ref 0 in
  for i = 0 to n-1 do
    s := !s + a.(i) * b.(i)
  done ;
  !s
```
2. 

```
let liste_vers_tableau l = match l with
| [] -> [|]
| x::_ -> (* pour avoir un x du bon type... *)
  let n = longueur l in
  let t = Array.make n x in (* ... on en a besoin ici *)
  let rec remplir_t l i = match l with
  | [] -> ()
  | y::q -> t.(i) <- y ; remplir_t q (i+1)
  in remplir_t l 0 ;
  t
```
3. 

```
let tableau_vers_liste t =
  let n = Array.length t in
  let rec creer_liste i =
    if i = n then []
    else t.(i) :: creer_liste (i+1)
  in creer_liste 0
```

```
4. let test_option f t =
    let n = Array.length t in
    let t' = Array.make n None in
    for i = 0 to n-1 do
        let x = t.(i) in
        if f x then t'.(i) <- Some x
    done ;
    t'
```

```
5. let premier_indice f t =
    let n = Array.length t in
    try
        for i = 0 to n-1 do
            if f t.(i) then raise (Trouve i)
        done ;
        None
    with Trouve i -> Some i
```

**Exercice 3 (typage) :** Pour chacune des fonctions suivantes :

- si elle est acceptée par OCaml, indiquer son type ;
- sinon, expliquer pourquoi elle est refusée.

1. fonction1 : `float array -> int -> float`
2. fonction2 : `(int -> float) -> int -> float`
3. fonction3 : `((('a -> 'b) -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c)`

**Exercice 4 (d'après Baccalauréat NSI – sujet 0) :**

1. Ce code sert à vider le contenu de `p` dans `q`. Ainsi, après exécution de ce code, `p` est vide, et `q` contient les anciens éléments de `p` mais à l'envers :

8
5
2
4

2. Le code de la question 1 permet de parcourir tous les éléments de `p` tout en les sauvegardant dans `q`, mais il faut ensuite les remettre dans `p`.

```
let hauteur_pile p =
    let q = creer_pile () in
    let n = ref 0 in
    while not (pile_vide p) do
        empiler q (depiler p) ;
        incr n
    done ;
    while not (pile_vide q) do
        empiler p (depiler q)
    done ;
    !n
```

3. Dans le même esprit, on parcourt les  $i$  premiers éléments de  $p$  en les sauvegardant dans  $q$  pour chercher le max, puis on remet ces éléments dans  $p$ .

```
let max_pile p i =  
  let q = creer_pile () in  
  let j = ref 1 in  
  let x_max = ref (sommet p) in  
  for k = 1 to i do  
    let x = depiler p in  
    empiler q x ;  
    if x > !x_max then  
      begin  
        x_max := x ;  
        j := k  
      end  
    end  
  done ;  
  while not (pile_vide q) do  
    empiler p (depiler q)  
  done ;  
  !j
```

4. L'astuce consiste à utiliser deux piles auxiliaires : on transfère d'abord les  $j$  premiers éléments de  $p$  dans  $q$ , puis on vide  $q$  dans  $q'$ , et enfin  $q'$  dans  $p$ .

```
let retourner p j =  
  (* on transfère les j premiers éléments de p dans q *)  
  let q = creer_pile () in  
  for k = 1 to j do  
    empiler q (depiler p) ;  
  done ;  
  (* on transfère les éléments de q dans q', en les retournant *)  
  let q' = creer_pile () in  
  while not (pile_vide q) do  
    empiler q' (depiler q) ;  
  done ;  
  (* on transfère les éléments de q' dans p *)  
  while not (pile_vide q') do  
    empiler p (depiler q') ;  
  done
```

5. 

```
let tri_crepes p =  
  let n = hauteur_pile p in  
  for i = n downto 1 do  
    let j = max_pile p i in  
    retourner p j ;  
    retourner p i  
  done
```

**Exercice 5 (Vendredi 13) :** Dans le code ci-dessous :

- le type `jour` représente les jours de la semaine ;
- les dates sont représentées par des triplets (*jour, mois, année*) ;
- la fonction `prochain_jour` renvoie le jour du lendemain ;
- la fonction `prochaine_date` renvoie la date du lendemain ;
- la fonction `loop` énumère tous les jours entre le 1<sup>er</sup> janvier 2024 et le 31 décembre 2099, et incrémente son argument `nb` à chaque fois qu'on rencontre un vendredi 13.

Ainsi, à la fin de l'exécution de ce programme, la variable `nb_vendredi_13` contient la valeur recherchée.

```
type jour = Lundi | Mardi | Mercredi | Jeudi | Vendredi | Samedi | Dimanche

let prochain_jour j = match j with
| Lundi    -> Mardi
| Mardi    -> Mercredi
| Mercredi -> Jeudi
| Jeudi     -> Vendredi
| Vendredi -> Samedi
| Samedi   -> Dimanche
| Dimanche -> Lundi

let prochaine_date (j, m, a) = match m with
| 1 | 3 | 5 | 7 | 8 | 10 when j = 31 -> (1, m+1, a)
| 4 | 6 | 9 | 11 when j = 30 -> (1, m+1, a)
| 2 when j = 28 && a mod 4 <> 0 -> (1, 3, a)
| 2 when j = 29 -> (1, 3, a)
| 12 when j = 31 -> (1, 1, a+1)
| _ -> (j+1, m, a)

let nb_vendredi_13 =
  let rec loop j d nb =
    let j' = prochain_jour j in
    let d' = prochaine_date d in
    match d with
    | (1, 1, 2100) -> nb
    | (13, _, _) when j = Vendredi -> loop j' d' (nb+1)
    | _ -> loop j' d' nb
  in loop Lundi (1, 1, 2024) 0
```