```
(* 2.1 *)
let curryfie f = fun x y -> f (x,y);;
let decurryfie f = fun (x,y) -> f x y;;
let plus_currifiee x y = x+y;;
let plus_decurrifiee (x,y) = x+y;;
(curryfie plus_decurrifiee) 2 3;; (* 5 *)
(decurryfie plus_currifiee) (2,3);; (* 5 *)

(* 2.2 *)
let compose f g = fun x -> f (g x);;
let fois_2 = ( prefix * ) 2;;
let fois_2_plus_1 = compose pred (compose fois_2 succ);;
(* succ est la fonction x -> x+1. pred est x -> x-1 *)
fois_2_plus_1 3;; (* 7 *)

let divise x y =
  y mod x = 0;;
#infix "divise";;
if 5 divise 10 then "ouf" else "bug";; (* "ouf" *)

let o f g x = f (g x);;
#infix "o";;
let fois_2_plus_1 = pred o fois_2 o succ;;
fois_2_plus_1 4;; (* 9 *)

(* 2.3 *)
let rec iter f = function
  | [] -> ()
  | t::q -> f t; iter f q;;
(* let iter = do_list *)
let print_int_list = iter print_int;;
(* ou bien let print_int_list = iter (fun i -> print_int i; print_char ',');; *)
print_int_list [1;2;3];; (* renvoie (), après avoir affiché 123 *)

(* 3.1 *)
let rev liste = it_list (fun list elt -> elt::list) [] liste;;
rev [1;2;3;4];; (* [4; 3; 2; 1] *)
let rev_map f liste = it_list (fun list elt -> f elt::list) [] liste;;
rev_map succ [1;2;3;4];; (* [5; 4; 3; 2] *)

let map f liste = list_it (fun elt list -> f elt :: list) liste [];;
map succ [1;2;3];; (* [2; 3; 4] *)
let map_option f liste =
  list_it
    (fun elt list ->
       match f elt with
         | None -> list
         | Some e -> e :: list)
    liste [];;
let pair x =
  x mod 2 = 0;;
map_option (fun x -> if pair x then Some (x/2) else None) [0;1;2;3;4;5];; (* [0; 1; 2] *)

(* 3.2 *)
let aiguillage liste = it_list (fun (la,lb) elt -> lb,elt::la) ([],[]) liste;;
(* Note: on peut même le faire en 25 tokens, mais il y a des appels de fonction en plus *)
(* let aiguillage liste = it_list (fun accu elt -> snd accu, elt::fst accu) ([],[]) liste;; *)
aiguillage [1;2;3;4;5;6;7;8;9];;
(* [8; 6; 4; 2], [9; 7; 5; 3; 1] *)

(* 3.3 *)
let rec fusion ordre la lb = match (la,lb) with
  | [],_ -> lb
  | _,[] -> la
  | ta::qa, tb::qb ->
      if ordre ta tb
      then ta :: fusion ordre qa lb
      else tb :: fusion ordre la qb;;
fusion (prefix <) [1;3;5] [2;2;2;9];;
(* [1; 2; 2; 2; 3; 5; 9] *)

(* 3.4 *)
let tri ordre liste =
  let rec fusionne_2_par_2 list = match list with
    | [] | _::[] -> list
    | a::b::q -> fusion ordre a b :: fusionne_2_par_2 q in
  let rec fusionne_tout = function
    | [] -> failwith "fusionne_tout"
    | [a] -> a
    | l -> fusionne_tout (fusionne_2_par_2 l) in
  match liste with
    | [] -> []
    | l -> fusionne_tout (map (fun x -> [x]) liste);;

let tri_croissant = tri (fun x y -> x < y);;
```

```
let tri_decroissant = tri (fun x y -> x > y);;
let l = [10;2;5;1;7;6;5;9] in tri_croissant l, tri_decroissant l;;
(* [1; 2; 5; 5; 6; 7; 9; 10], [10; 9; 7; 6; 5; 5; 2; 1] *)

let lowercase c = char_of_int (int_of_char c +32);;
let string_lowercase s =
  let len = string_length s in
  let result = create_string len in
  for i=0 to pred len do
    let c = s.[i] in
    result.[i] <-
      (if 'A' <= c && c <= 'Z'
       then lowercase c
       else c)
  done;
  result;;

(* 3.5 *)
let tri_case_insensitive =
  tri (fun x y -> string_lowercase x < string_lowercase y);;
tri_case_insensitive ["foo"; "bAr"; "FOO"; "bar"; "fOO"]
(* ["bar"; "bAr"; "fOO"; "FOO"; "foo"] *)
(*note: l'ordre entre les mots identiques aux majuscules près n'est pas spécifié.*)

let tri ordre liste =
  it_list (fun la lb -> fusion ordre la lb) [] (map (fun x -> [x]) liste);;
(* C'est le tri par insertion,
   car à chaque étape de it_list on fusionne l'accumulateur avec une liste à un élément. *)
tri (prefix <) [6;3;4;2;2;8;5;4];;
(* [2; 2; 3; 4; 4; 5; 6; 8] *)
```