```
(** Ex1 *)
let affiche_liste_poly aff l = (* fonction polymorphe d'affichage d'une liste à
partir d'une fonction d'affichage d'un élément aff *)
  let stdaff = fun out -> aff in
  Printf.printf "[";
  begin match l with
  | [] -> ()
  | h::t -> Printf.printf "%a" stdaff h;
            List.iter (Printf.printf "; %a" stdaff) t
  end;
  Printf.printf "]"

let affiche_liste_entiers l = List.iter (fun x -> print_int x; print_char ' ') l

let count x l = List.fold_left (fun acc y -> if x=y then acc+1 else acc) 0 l

let flatten l = List.fold_right (fun a b -> List.append a b) l []

(* fst源代码:
   external fst : 'a * 'b -> 'a = "%field0" *)
let fst_list l = List.map fst l;;

let fst_list_right l = List.fold_right (fun a b -> let (x,y) = a in x::b) l []

let fst_list_left l = List.fold_left (fun b a-> let (x,y) = a in x::b) [] l

(** Ex2 *)
(* CRÉATION D'UNE LISTE QUELCONQUE *)
let _ = Random.self_init ()

let rec rdm_int_list bound len acc =
(* ajout d'une liste de taille len de nombres aléatoires entre 0 et bound (exclu)
à la liste acc *)
  if len = 0
  then acc
  else let ne = Random.int bound in
       rdm_int_list bound (len-1) (ne::acc)

let make_list n = rdm_int_list (n-1) n []

let couper_1 l =
    let rec aux l1 l2 l =
        match l with
        | [] -> l1,l2
        | [x] -> x::l1,l2
        | x::y::l -> aux (x::l1) (y::l2) l
    in
aux [] [] l

let rec couper1_bis l = match l with
  | [] | [_] -> l, []
  | h1::h2::t -> let l1, l2 = couper1_bis t in
                 h1::l1, h2::l2

let couper_2 l =
    let rec aux b l1 l2 l =
        match l with
        | [] -> l1,l2
        | x::s -> if b then aux (not b) (x::l1) l2 s
                  else aux (not b) l1 (x::l2) s
    in
aux true [] [] l

let couper2_bis l =
  let rec couperrec ((l1, l2) as acc) = function
  | [] -> acc
  | h::t -> couperrec (l2, h::l1) t in
  couperrec ([], [])  l

let couper_3 l =
    List.fold_left
    (fun (b,(l1,l2)) x -> not b, if b then (x::l1,l2) else (l1,x::l2))
    (true,([],[])) l
|> snd

let couper3_bis l =
  List.fold_left (fun (l1, l2) h -> l2, h::l1) ([], []) l

(* comp x y = 0 si x=y
   comp x y < 0 si x<y
   comp x y > 0 si x>y *)
let rec fusion comp l1 l2 =
    match l1,l2 with
    | _,[] -> l1
    | [],_ -> l2
    | s1::r1,s2::r2 -> if comp s1 s2 < 0 then s1::(fusion comp r1 l2)
                                         else s2::(fusion comp l1 r2)

let rec fusion_bis comp l1 l2 =
  match l1, l2 with
  | [], l | l, [] -> l
  | h1::t1, h2::t2 ->
     if comp h1 h2 <= 0
     then h1 :: fusion_bis comp t1 l2
     else h2 :: fusion_bis comp l1 t2

let fusion_rt comp l1 l2 =   (* version récursive terminale avec accumulateur *)
  let rec fusionrec l1 l2 acc =
    match l1, l2 with
    | [], l | l, [] ->
       List.rev_append acc l (* attention à n'utiliser ni List.rev ni List.append
ici *)
    | h1::t1, h2::t2 ->
       if comp h1 h2 <= 0
       then fusionrec t1 l2 (h1 :: acc)
       else fusionrec l1 t2 (h2 :: acc) in
  fusionrec l1 l2 []

let rec trier comp l =
    match l with
    | [] -> []
    | [x] -> [x]
    | _ -> let (l1,l2) = couper_3 l in
           fusion comp (trier comp l1) (trier comp l2)

let rec trier_cps comp l cont = (* version récursive terminale avec continuation *)
  match l with
  | [] | [_] -> cont l
  | _ -> let l1, l2 = couper3_bis l in (* couper3 est récursive terminale *)
         trier_cps comp l1 (fun tl1 ->
         trier_cps comp l2 (fun tl2 ->
         cont (fusion_rt comp tl1 tl2)))

let numerotation l =
  List.fold_left (fun (accl, accn) h -> (* l'accumulateur contient aussi le numéro
à donner au prochain élément *)
      (h, accn)::accl, accn + 1) ([], 1) l
  |> fst
  |> List.rev (* on a utilisé fold_left, il faut donc remettre les éléments dans
le bon ordre *)
```

```
let comp_assoc (a1, a2) (b1, b2) =
  compare a1 b1

let affiche_liste = affiche_liste_poly (fun (a,b) -> Printf.printf "(%d, %d)" a b)

let numerote_trie_affiche tri l =
  numerotation l
  |> tri comp_assoc
  |> affiche_liste

let couper4 l =
  let n = List.length l in
  let rec take k l =
    if k = 0
    then [], l
    else match l with
         | [] -> assert false
         | h::t -> let l1, l2 = take (k-1) t in
                   h::l1, l2 in
  take (n/2) l
```