

(* Question 1. *)

```
let decomposer_entier n =  
  let rec aux n n2 acc =  
    match n, n2 with  
    | l, l2 when l2 = 0 → acc @ [(l, l2)]  
    | l, l2 → acc @ [(l, l2)] @ aux (l+1) (l2-1) acc  
  in aux 0 n [];;
```

(* Question 2. *)

```
let ajouter_noeud_unaire arbres =  
  List.map (fun id → Unaire(id)) arbres;;
```

(* Question 3. *)

```
let ajouter_noeud_binaire arbres1 arbres2 =  
  let rec aux a a2 acc =  
    match a, a2 with  
    | [], _ → acc  
    | h::t, [] → acc @ aux t arbres2 acc  
    | h::t, h'::t' → acc @ [Binaire (h, h')] @ aux a (t') acc  
  in aux arbres1 arbres2 [];;
```

(* Question 4. *)

```
let taille arb =  
  let rec aux arb count =  
    match arb with  
    | Binaire(a, b) → count + aux a 1 + aux b 1  
    | Unaire(a) → count + aux a 1  
    | Feuille(v) → 0  
  in aux arb 1;;
```

```
;;
```

(* Question 5. *)

```
let rec generer_arbres v n =  
  if n = 0 then [Feuille v]  
  else  
    let lst1 = ajouter_noeud_unaire (generer_arbres v (n - 1))  
    and decomp = decomposer_entier (n-1) in  
    let lst2 = List.fold_left (fun acc (a, b) → acc @  
      (ajouter_noeud_binaire (generer_arbres v a) (generer_arbres v b))) [] decomp  
    in lst1 @ lst2;;
```