```
(*************************************************************************)
(*                          Partitions d'entiers                         *)
(*************************************************************************)

(* Afficher une liste dans l'ordre... *)

let rec affiche_liste = function
    []   -> print_newline ()
  | a::l -> print_int a ; print_string " " ; affiche_liste l
;;

(* ...et dans l'ordre inverse. *)

let affiche_rev l = aff_rec l ; print_newline ()
  where rec aff_rec = function
    []   -> ()
  | a::l -> aff_rec l ; print_int a ; print_string " "
;;

(* Afficher les partitions de n en parts <= m dans l'ordre decroissant : *)

let partitions_m n m = parts_rec [] m n
  where rec parts_rec l m = function
    0 -> affiche_rev l
  | n -> for i = m downto 1 do
           if i<=n then parts_rec (i::l) i (n-i)
         done
;;

(* ...d'ou les partitions de n : *)

let partitions n = partitions_m n n;;

(* calcul de p_m(n) et de p(n) = p_n(n) : *)

let rec p_m_n = fun
    _ 0 -> 1
  | 1 _ -> 1
  | m n -> if m > n then
           p_m_n (pred m) n
         else
           (p_m_n (pred m) n) + (p_m_n m (n-m))
;;

let p_n n = p_m_n n n;;

(* Soit A_n l'ensemble des partitions de n en au plus m parts,
 * et   B_n l'ensemble des partitions de n en parts au plus egales a m.
 *
 * Soit n=a1+...+as une partition de A_n. En regroupant les ai egaux, on
 * peut ecrire cette partition comme la somme des a'i x ni (ou les a'i sont
 * distincts et {ai}={a'i}). on a s <= m, donc ni <= m. Et donc les ni
 * forme une partition de B_n. On a donc une surjection de B_n dans A_n.
 *
 * Reciproquement, soit n=n1+...+ns une partition de B_n (donc ni <= m).
 * Soit ai le nombre de ni egaux a i. Alors (a1, 2xa2, 3xa3, ..., mxam)
 * est une partition de n en au plus m parts, (ie) une partition de A_n.
 * On a donc une surjection de A_n dans B_n.
 *
 * Les deux ensembles A_n et B_n etant finis, leurs deux cardinaux
 * sont donc egaux par double inegalite. *)

(* Afficher les partitions de n en parts distinctes.
 * Comme precedemment, mais lorsque l'on prend la part i on se rappelle
 * recursivement avec (i-1) au lieu de i. *)

let parts_distinctes n = parts_rec [] n n
  where rec parts_rec l m = function
    0 -> affiche_rev l
  | n -> if n>0 then
           for i = m downto 1 do
             if i<=n then parts_rec (i::l) (i-1) (n-i)
           done
;;

(* Afficher les partitions de n en parts distinctes et impaires.
 * Comme precedemment mais on teste le caractere impair de i. *)

let impair n = (n mod 2) <> 0
;;

let parts_distinctes_impaires n = parts_rec [] n n
  where rec parts_rec l m = function
    0 -> affiche_rev l
  | n -> if n>0 then
           for i = m downto 1 do
             if i<=n & (impair i) then parts_rec (i::l) (i-1) (n-i)
           done
;;

(* Parts dans une liste donnee : *)

let parts_dans_l n liste = parts_rec [] liste n
  where rec parts_rec l liste = function
    0 -> affiche_rev l
  | n -> if n>0 then begin
           match liste with
             []   -> ()
           | i::r -> if i<=n then parts_rec (i::l) liste (n-i) ;
                     parts_rec l r n
         end
;;

(* Parts dans une liste donnee avec ressources :
 * comme precedemment, mais chaque fois que l'on utilise une part i
 * alors on decremente sa ressource ni.
 * Et lorsqu'une ressource ni arrive a 0 alors on poursuit avec les
 * ressources suivantes. *)

let monnaie n liste = parts_rec [] liste n
  where rec parts_rec l liste = function
    0 -> affiche_rev l
  | n -> if n>0 then begin
           match liste with
             []        -> ()
           | (i,0) ::r -> parts_rec l r n
           | (i,ni)::r -> if i<=n then parts_rec (i::l) ((i,ni-1)::r) (n-i) ;
                          parts_rec l r n
         end
;;
```