```
(****** Algorithmes combinatoires et monades ********)

module type FONCTEUR =
  sig
    type 'a t
    val map : ('a -> 'b) -> ('a t -> 'b t)
  end

module type MONADE =
  sig
    include FONCTEUR
    val return : 'a -> 'a t
    val (>>=)  : 'a t -> ('a -> 'b t) -> 'b t
  end

module type MONADE_PLUS =
  sig
    include MONADE
    val zero : 'a t
    val (++) : 'a t -> 'a t -> 'a t
  end

(* interface incluant l'affichage des éléments calculés *)
(* pour les listes d'entiers uniquement                 *)
module type MONADE_PLUS_PRINT =
  sig
    include MONADE_PLUS
    val print : Format.formatter -> int list t -> unit
  end

(* fonction auxiliaire pour compter le nombre maximum d'octets alloués en mémoire *)
let max_bytes () =
  let stat = Gc.stat () in
  8. *. (stat.minor_words +. stat.major_words -. stat.promoted_words)

(* fonction auxiliaire pour afficher une liste d'entiers *)
let print_int_list fmt l =
  begin
    Format.fprintf fmt "[";
    List.iter (Format.fprintf fmt "%d; ") l;
    Format.fprintf fmt "]"
  end


(* implantation de la monade ND avec des listes *)
(* ne fonctionne qu'en l'absence de doublons    *)
module NDList : MONADE_PLUS_PRINT =
  struct
    type 'a t = 'a list
    let map = List.map
    let return v = [v]
    let (>>=) s f = List.flatten (List.map f s)
    let zero = []
    let (++) = (@)

    (* fonction d'affichage pour les tests *)
    let print fmt =
      List.iter (Format.fprintf fmt "%a@." print_int_list)
  end


(*** Combinaisons d'une liste ***)

module Exo1 (ND : MONADE_PLUS) =
  struct
    (* CONTRAT
       Fonction qui renvoie toutes les combinaisons possible de k éléments d'une liste l
       Paramètre k : le nombre d'éléments dans la liste retournée
       Précondition : k>= taille de l
       Paramètre l : la liste dans laquelle on prend les éléments
       Résultat : les combinaisons de k éléments choisis dans l
     *)
    let rec combinaisons k l =
      match k , l with
      | 0,_     -> ND.return []
      | _, []   -> ND.zero
      | _, t::q -> ND.(combinaisons k q ++ (combinaisons (k-1) q >>= fun combi -> return (t::combi)))

  end

(* TESTS *)
let test1 (module ND : MONADE_PLUS_PRINT) =
  let module M = Exo1 (ND) in
  let old_bytes = max_bytes () in
  let result = M.combinaisons 4 [1;2;3;4;5;6;7;8;9;10] in
  begin
    Format.printf "@.TEST combinaisons@.memory used: %f@.result:@. %a@." (max_bytes () -. old_bytes) ND.print result
  end

let _ = test1 (module NDList)


(*** Permutations d'une liste ***)

module Exo2 (ND : MONADE_PLUS) =
  struct
    (* CONTRAT
       Fonction prend en paramètre un élément e et une liste l et qui insére e à toutes les possitions possibles dans l
       Pamaètre e : ('a) l'élément à insérer
       Paramètre l : ('a l  ist) la liste initiale dans laquelle insérer e
       Résultat : toutes les insertions possible de e dans l
     *)

    let rec insertion e l =
      match l with
      |[]   -> ND.return [e]
      |t::q -> ND.(return (e::l) ++ (insertion e q >>= fun ins -> return (t::ins)))

    (* CONTRAT
       Fonction qui renvoie la liste des permutations d'une liste
       Paramètre l : une liste
       Résultat : les permutations de l (toutes différentes si les élements de l sont différents deux à deux)
```

```
     *)
    let rec permutations l =
      match l with
      | []   -> ND.return []
      | t::q -> ND.(permutations q >>= fun perm -> insertion t perm)
end

(* TESTS *)
let test2 (module ND : MONADE_PLUS_PRINT) =
  let module M = Exo2 (ND) in
  let old_bytes = max_bytes () in
  let result = M.permutations [1;2;3;4;5] in
  begin
    Format.printf "@.TEST permutations@.memory used: %f@.result:@. %a@."
(max_bytes () -. old_bytes) ND.print result
  end

let _ = test2 (module NDList)


(*** Partition d'un entier ***)

module Exo3 (ND : MONADE_PLUS) =
  struct
    (* CONTRAT
       partition int -> int list
       Fonction qui calcule toutes les partitions possibles d'un entier n
       Paramètre n : un entier dont on veut calculer les partitions
       Préconditions : n >0
       Résultat : les partitions de n
     *)
    let partitions n =
      let rec partitions_aux n t =
        if      t > n then ND.zero
        else if t = n then ND.return [t]
        else (* t < n *)   ND.((partitions_aux (n-t) t >>= fun part -> return
(t::part)) ++ partitions_aux n (t+1))
      in partitions_aux n 1
  end

(* TESTS *)
let test3 (module ND : MONADE_PLUS_PRINT) =
  let module M = Exo3 (ND) in
  let old_bytes = max_bytes () in
  let result = M.partitions 5 in
  begin
    Format.printf "@.TEST partitions@.memory used: %f@.result: %a@."
(max_bytes () -. old_bytes) ND.print result
  end

let _ = test3 (module NDList)


(* fonction auxiliaire pour réaliser tous les tests des fonctions
combinatoires *)
let test_combinatoire (module ND : MONADE_PLUS_PRINT) =
  begin
    test1 (module ND);
    test2 (module ND);
    test3 (module ND)
  end


(*** Autre implantation de ND par itérateurs ***)

module NDIter : MONADE_PLUS_PRINT =
  struct
    type 'a t = Tick of ('a * 'a t) option Lazy.t

    let next (Tick it) = Lazy.force it

    let rec map f (it : 'a t) =
      Tick (lazy (
                match next it with
                | None          -> None
                | Some (a, it') -> Some (f a, map f it')
        ))

    let return a =
      Tick (lazy (Some (a, Tick (lazy None))))

    let zero =
      Tick (lazy None)

    let rec (++) it1 it2 =
      Tick (lazy (
                match next it1 with
                | None           -> next it2
                | Some (a1, it1') -> Some (a1, it1' ++ it2)
        ))

    let rec (>>=) it f =
      Tick (lazy (
                match next it with
                | None -> None
                | Some (a, it') -> next (f a ++ (it' >>= f))
        ))

    (* fonction d'affichage pour les tests *)
    let rec print fmt it =
      match next it with
      | None          -> Format.fprintf fmt "@."
      | Some (a, it') -> Format.fprintf fmt "%a@.%a" print_int_list a print
it'
  end

(* TESTS *)
let test_iter () =
  begin
    Format.printf "@.TEST itérateur@.";
    test_combinatoire (module NDIter)
  end


(*** Autre implantation de ND par tirage aléatoire ***)

module NDRandom : MONADE_PLUS_PRINT =
  struct
    type 'a t = unit -> 'a option

```

```
    let map f it =
      fun () -> match it () with
                | None   -> None
                | Some a -> Some (f a)

    let return a = fun () -> Some a

    let (>>=) it f =
      fun () -> match it () with
                | None   -> None
                | Some a -> f a ()

    let zero = fun () -> None

    let (++) it1 it2 =
      fun () -> if Random.bool ()
                then
                  match it1 () with
                  | None -> it2 ()
                  | r    -> r
                else
                  match it2 () with
                  | None -> it1 ()
                  | r    -> r

    (* fonction d'affichage pour les tests *)
    let print fmt it =
      match it () with
      | None   -> Format.fprintf fmt "@."
      | Some v -> Format.fprintf fmt "%a@." print_int_list v
  end

(* TESTS *)
let test_random (module ND : MONADE_PLUS_PRINT) =
  begin
    Format.printf "@.TEST aléatoire@."; test_combinatoire (module ND)
  end

let _ = test_random (module NDRandom)

(*** Autre implantation de ND par tirage aléatoire équitable ***)

module NDFairRandom : MONADE_PLUS_PRINT =
  struct
    type 'a t = int * (unit -> 'a)

    let map f (n, it) = (n, fun () -> f (it ()))

    let return a = (1, fun () -> a)

    let zero = (0, fun () -> raise Not_found)

    let (++) (n1, it1) (n2, it2) =
      match n1, n2 with
      | 0, 0 -> zero
      | 0, _ -> (n2, it2)
      | _, 0 -> (n1, it1)
      | _    -> (n1+n2, fun () -> if Random.int (n1+n2) < n1 then it1 () else
it2 ())

    let (>>=) (n, it) f =
      match n with
      | 0 -> zero
      | _ -> match f (it ()) with
             | (p, _) -> (n*p, fun () -> snd (f (it ())) ())

    (* fonction d'affichage pour les tests *)
    let print fmt (n, it) =
      match n with
      | 0 -> Format.fprintf fmt "@."
      | _ -> Format.fprintf fmt "%a@." print_int_list (it ())
  end

(* TESTS *)
let _ = test_random (module NDFairRandom)

```