TD16.ml

```
(* Dans les deux types suivants, 'a correspond à la structure
   utilisée pour représenter une solution partielle au
   problème. Pour l'exemple des n reines, on peut imaginer
   plusieurs possibilités :
   - une liste contenant les positions des reines déjà fixées ;
   - un tableau n * n représentant l'échiquier, avec une certaine valeur
     pour les cases occupées par les reines, une pour les cases
     interdites, une pour les cases libres. Dans ce cas,
     on voudra sans doute aussi stocker un entier donnant le nombre
     de reines déjà placées.
   - un tableau de taille n donnant les positions des reines, avec par
     exemples des 0 pour celles qui ne sont pas encore placées. Là aussi,
     on préférera sans doute noter aussi le nombre de reines placées.
   - beaucoup d'autres choses plus ou moins raisonnables sont imaginables.
*)

type 'a reponse =
  | Refus
  | Accepte of 'a
  | Partiel of 'a

type 'a probleme =
  {accepte : 'a -> 'a reponse;
   enfants : 'a -> 'a list;
   initiale : 'a}

let resout {accepte; enfants; initiale} =
  let rec backtrack position =
  match accepte position with
  | Refus -> None
  | Accepte solution -> Some solution
  | Partiel candidat ->
     let u = enfants position in
     explore u
  and explore u =
    match u with
    | [] -> None
    | x :: xs ->
       match backtrack x with
         None -> explore xs
       | Some solution -> Some solution in
  backtrack initiale


(*****************************************************************************)
(*                  Problème des n reines                                    *)
(*****************************************************************************)


let rec range a b = if a >= b then [] else a :: range (a + 1) b

let enfants_reines (tab, k) =
  let n = Array.length tab in
  let f i =
    let t = Array.copy tab in
    t.(k) <- i;
    (t, k + 1) in
  List.map f (range 1 (n + 1))

exception Echec

let accepte_reines (tab, k) =
  try
    for i = 0 to k - 2 do
      if tab.(i) = tab.(k - 1) || abs (tab.(i) - tab.(k - 1)) = k - 1 - i then
        raise Echec
    done;
    if k = Array.length tab then
      Accepte (tab, k)
    else
      Partiel (tab, k)
  with
    Echec -> Refus

let nreines n = {enfants = enfants_reines;
                 accepte = accepte_reines;
                 initiale = (Array.make n 0, 0)}


(*
utop[123]> resout (nreines 24);;
- : (int array * int) option =
Some
 ([|1; 3; 5; 2; 4; 9; 11; 14; 18; 22; 19; 23; 20; 24; 10; 21; 6; 8; 12; 16;
    13; 7; 17; 15|],
  24)
*)


(* Les deux variantes très simples de la fonction resout,
   permettant respectivement d'obtenir la liste de toutes
   les solutions et le nombre de solutions. *)
let enumere problem =
  let rec backtrack position =
  match problem.accepte position with
  | Refus -> []
  | Accepte solution -> [solution]
  | Partiel candidat ->
     let u = problem.enfants position in
     List.concat (List.map backtrack u) in
  backtrack problem.initiale

let compte problem =
  let rec backtrack position =
    match problem.accepte position with
    | Refus -> 0
    | Accepte solution -> 1
    | Partiel candidat ->
       let u = problem.enfants position in
       List.fold_left (+) 0 (List.map backtrack u) in
  backtrack problem.initiale


(*
utop[122]> compte (nreines 12);;
- : int = 14200
 *)


(*****************************************************************************)
(*                  Tableaux auto-référents                                  *)
(*****************************************************************************)


let occurrences t n =
  let occs = Array.make n 0 in
  Array.iter (fun x -> occs.(x) <- occs.(x) + 1) t;
  occs
```

TD16.ml

```
let accepte_auto n t =
  if Array.length t = n then
    let occs = occurrences t n in
    if occs = t then Accepte t
    else Refus
  else Partiel t

let enfants_auto n t =
  let f i = Array.append t [| i |] in
  List.map f (range 0 n)

let autoreferent_brute n =
  {accepte = accepte_auto n;
   enfants = enfants_auto n ;
   initiale = [| |]}




(* Pour accélérer la recherche, il faut élaguer l'arbre (repérer le plus
   rapidement possible qu'on se trouve dans une branche ne pouvant donner
   de solution).
   On utilise les remarques suivantes :
   - à la fin, la somme doit faire n. Si on dépasse n, c'est donc perdu.
   - c'est également perdu si l'on est sûr qu'on dépassera n plus tard.
   - s'il y a déjà plus d'occurrences d'une valeur i que le contenu de
     t.(i), c'est également perdu (le contenu de t.(i) ne changera plus,
     le nombre d'occurrences ne peut que croître).
   - inversement, si par exemple t.(3) vaut 5 et qu'il n'y a qu'un seul
     3 dans le tableau pour l'instant, alors il faut réserver 4 cases
     futures pour y mettre des 3. Si ce n'est pas possible, c'est
     perdu.
   - On pourrait encore clairement améliorer, mais cela suffit pour se
     convaincre qu'il existe une unique solution (sauf pour certaines
     petites valeurs de n). Reste à le prouver...
 *)
let accepte_auto_bis n t =
  let k = Array.length t in
  if n = k then accepte_auto n t
  else
    try
      let somme = Array.fold_left (+) 0 t in
      if somme > n then raise Echec;
      let occs = occurrences t n in
      if k > 0 && somme + (n - k) - (t.(0) - occs.(0)) > n then raise Echec;
      let dispo = ref (n - k) in
      for i = 0 to k - 1 do
        dispo := !dispo - t.(i) + occs.(i);
        if occs.(i) > t.(i) || !dispo < 0 then raise Echec;
      done;
      Partiel t
    with
      Echec -> Refus


let autoreferent_opt n =
  {enfants = enfants_auto n ;
   accepte = accepte_auto_bis n ;
   initiale = [| |]}
```