Correction du DS 5

Julien Reichert

La correction des questions de cours étant dans le cours, elle ne sera pas donnée ici. De même, les exercices issus des TD et TP ont leur correction déjà publiée.

Exercices

Exercice 1 :

```
bool exo1(char* chaine)
{
  int actuel = 0; int maximal = 0;
  for (int indice = 0 ; chaine[indice] != '\0' ; indice += 1)
  {
    if (chaine[indice] == '1')
    {
      actuel += 1;
      if (actuel > maximal)
      {
        maximal = actuel;
        if (maximal > 3) return false;
      }
    }
    else actuel = 0;
  }
  return maximal == 3;
}
```

Exercice 2 :

```
bool exo2(char* chaine)
{
  int actuel0 = 0; int maximal0 = 0; int actuel1 = 0; int maximal1 = 0;
  for (int indice = 0 ; chaine[indice] != '\0' ; indice += 1)
  {
    if (chaine[indice] == '1')
    {
      actuel0 = 0; actuel1 += 1;
      if (actuel1 > maximal1) maximal1 = actuel1;
    }
    else
    {
      actuel1 = 0; actuel0 += 1;
      if (actuel0 > maximal0) maximal0 = actuel0;
    }
  }
  return maximal0 == 3 || maximal1 == 3;
}
```

Exercice 3 :

Aucun piège, on peut remplacer le `||` dans le `return` par un `&&`, sachant qu'il est imaginable d'interrompre la fonction prématurément si un des maximums dépasse trois.

Exercice 4 :

Une possibilité est de donner pour toute taille d'un préfixe de `w` l'ensemble des tailles possibles des préfixes de `u` qui peuvent être incorporés jusqu'à l'indice en question dans `w`, la taille du préfixe restant dans `v` s'en déduisant.

```
exception Impossible

let est_shuffle u v w =
  let nu = String.length u in
  let nv = String.length v in
  let nw = String.length w in
  let memoire = Array.init (nw + 1) (fun _ -> Hashtbl.create 8) in
  Hashtbl.add memoire.(0) 0 true;
  try
    for i = 0 to nw-1 do
      Hashtbl.iter
      (
        fun ind _ ->
        if ind < nu && w.[i] = u.[ind] then Hashtbl.replace memoire.(i+1) (ind+1) true;
        if i-ind < nv && w.[i] = v.[i-ind] then Hashtbl.replace memoire.(i+1) (i-ind+1) true
      ) memoire.(i);
      if Hashtbl.length memoire.(i+1) = 0 then raise Impossible
    done; true (* équivalent à Hashtbl.mem memoire.(i+1) nu *)
  with Impossible -> false
```

Exercice 5 :

En admettant que les opérations sur les tables de hachage sont en temps constant (ou en travaillant sur une matrice au lieu d'un tableau de tables de hachage), le nombre de clés dans chaque table est majoré par `nu`, ce qui donne une complexité linéaire en la taille de `u` pour chaque tour de la boucle, d'où un total en $\mathcal{O}(\texttt{nu.nw})$.

Un algorithme d'exploration exhaustive aurait pu tester tous les découpages possibles, dont le nombre est $\binom{\texttt{nw}}{\texttt{nu}}$. C'est une tendance exponentielle qu'il fallait clairement éviter, même si l'essentiel des découpages sont rejetés rapidement, et en pratique si les deux mots n'ont pas de lettre en commun un algorithme naïf est tout aussi efficace que l'algorithme écrit ci-avant.

Exercice 6 :

```
(* type graphe_echecs = (string * int, (string * int, bool) Hashtbl.t) Hashtbl.t *)

let blanches_menacees_non_protegees echiquier =
  let blanc = Hashtbl.fold (fun (i, p) _ acc -> if p = -1 then (i, p)::acc else acc) echiquier [] in
  let accessible joueur (i, p) = Hashtbl.fold
  (fun (id, prop) dest boolacc -> boolacc || prop = joueur && Hashtbl.mem dest (i, p)) echiquier false in
  List.filter (fun (i, p) -> accessible 1 (i, p) && not (accessible (-1) (i, p))) blanc
```

Problème 1

Question P1.1 :

Considérons un système avec des pièces de 1, 12 et 20 et la somme de 16 à payer. Alors l'optimum revient à payer avec deux pièces de 20 et à se faire rendre deux pièces de 12, le dépassement étant alors de 24. On observe qu'aucune transaction faisant intervenir des pièces de valeur 1 ne peut être plus efficace, pour conclure que quatre pièces est l'optimum.

Question P1.2 :

On considère que le tableau décrivant le système est décroissant, conformément à l'énoncé.

```
int glouton_paiement_rendu(int* systeme, int taille, int s)
{
  int nb_pieces = s / systeme[0];
  int total = nb_pieces * systeme[0];
  int indice = 0;
  if (total == s) return nb_pieces;
  while (indice < taille && total + systeme[indice] > s)
  {
    indice += 1;
  }
  total += systeme[indice-1];
  nb_pieces += 1;
  while (total > s)
  {
    int nb_pieces_en_plus = (total - s) / systeme[indice];
    total -= nb_pieces_en_plus * systeme[indice];
    nb_pieces += nb_pieces_en_plus;
    indice += 1;
  }
  return nb_pieces;
}
```

Question P1.3 :

Avec le même exemple que dans la première question (optimum 4), l'algorithme commence par une division euclidienne :

```
nb_pieces : 0, total : 0, indice : 0
```

Par la suite, la première boucle conditionnelle s'arrête au premier tour et incorpore la pièce de 20 pour dépasser la somme.

```
nb_pieces : 1, total : 20, indice : 1
```

Dans la deuxième boucle conditionnelle, on commence par avancer l'indice sans rien retrancher au total, puis une fois que l'indice est celui de la pièce de 1, on retire quatre fois la valeur de cette pièce.

```
nb_pieces : 5, total : 16, indice : 3
```

On quitte alors la boucle et le résultat (non optimal) est renvoyé.

Question P1.4 :

Puisque les divisions euclidiennes interviennent à deux moments, la complexité est réduite à deux explorations partielles du système, qui ne se chevauchent en pratique pas. Ce sera donc linéaire en la taille du système quelle que soit la valeur à payer.

Question P1.5 :

```
let rendu_ids systeme somme =
  let rec ids prof_max =
    let rec dfs (paye, rendu) total prof =
      if total = somme then (paye, rendu)
      else if prof = prof_max then ([], [])
      else List.fold_left
      (
        fun solution valeur ->
        if solution <> ([], []) then solution
        else let payer = dfs (valeur::paye, rendu) (total + valeur) (prof+1) in
        if payer <> ([], []) then payer
        else dfs (paye, valeur::rendu) (total - valeur) (prof + 1)
      ) ([], []) systeme
    in let essai = dfs ([], []) 0 0 in
    if essai <> ([], []) then essai
    else ids (prof_max + 1)
  in if somme = 0 then ([], [])
  else ids 1;;
```

Question P1.6 :

Si un parcours en profondeur à profondeur maximale k échoue, on aura testé $(2n)^k$ possibilités (où n est le nombre de pièces, qu'on comptabilise pour payer et pour rembourser).

On peut limiter aux listes décroissantes de taille k pour éviter les répétitions, mais cela restera exponentiel en la valeur de la réponse. La programmation dynamique s'impose clairement.

Question P1.7 :

Voir la correction de l'exercice 10 du TP 7 avec `creer_file`, `enfiler_file`, `defiler_file` (utilisant la fonction auxiliaire extérieure `transfert`) et `est_vide_file`. **Attention aux signatures, la structure est persistante !**

Question P1.8 :

```
let paiement_optimal systeme s =
  let memoire = Hashtbl.create 8 in
  Hashtbl.add memoire 0 0;
  let rec mouline f =
    if est_vide_file f then failwith "Impossible de procéder au paiement !";
    let valeur, reste = defiler_file f in
    let profondeur = Hashtbl.find memoire valeur in
    if valeur = s then profondeur
    else let rec aux etat_file l = match l with (* On va arrêter les fold ici *)
    | [] -> etat_file
    | a::q ->
      let ef1 =
        if Hashtbl.mem memoire (valeur+a)
        then etat_file
        else (Hashtbl.add memoire (valeur+a) (profondeur+1); enfiler_file etat_file (valeur+a)) in
      let ef2 =
        if Hashtbl.mem memoire (valeur-a)
        then ef1
        else (Hashtbl.add memoire (valeur-a) (profondeur+1); enfiler_file ef1 (valeur-a))
      in aux ef2 q
    in mouline (aux reste systeme)
  in mouline (enfiler_file (creer_file ()) 0)
```

Question P1.9 :

Modification : remplacer les valeurs dans le dictionnaire par le couple de listes.

Le principe du parcours en largeur et l'utilisation d'une file garantissent la correction.

```
let paiement_optimal systeme s =
  let memoire = Hashtbl.create 8 in
  Hashtbl.add memoire 0 ([], []);
  let rec mouline f =
    if est_vide_file f then failwith "Impossible de procéder au paiement !";
    let valeur, reste = defiler_file f in
    let (paye, rendu) = Hashtbl.find memoire valeur in
    if valeur = s then (paye, rendu)
    else let rec aux etat_file l = match l with (* On va arrêter les fold ici *)
    | [] -> etat_file
    | a::q ->
      let ef1 =
        if Hashtbl.mem memoire (valeur+a)
        then etat_file
        else (Hashtbl.add memoire (valeur+a) (a::paye, rendu); enfiler_file etat_file (valeur+a)) in
      let ef2 =
        if Hashtbl.mem memoire (valeur-a)
        then ef1
        else (Hashtbl.add memoire (valeur-a) (paye, a::rendu); enfiler_file ef1 (valeur-a))
      in aux ef2 q
    in mouline (aux reste systeme)
  in mouline (enfiler_file (creer_file ()) 0)
```

Problème 2

Question P2.1 :

Il existe $n!$ permutations d'un ensemble de taille n.

Question P2.2 :

```
int factorielle(int n) // Dépasse certes vite du type int
{
  int rep = 1;
  for (int i = 2 ; i <= n ; i += 1) rep *= i;
  return rep;
}

int** engendre_peen(int n)
{
  assert (n > 0);
  if (n == 1) // Plus simple que 2 comme cas de base.
  {
    int** resultat = malloc(sizeof(int*));
    resultat[0] = malloc(sizeof(int));
    resultat[0][0] = 0;
    return resultat;
  }
  int** tableau = engendre_peen(n-1);
  int taille = factorielle(n-1);
  int taille2 = n * taille;
  int** resultat = malloc(taille2 * sizeof(int*));
  for (int j = 0 ; j < taille ; j += 1)
  {
    for (int i = 0 ; i < n ; i += 1)
    {
      resultat[j*n + i] = malloc(n * sizeof(int));
      for (int k = 0 ; k < i ; k += 1)
      {
        resultat[j*n + i][k] = tableau[j][k];
      }
      resultat[j*n + i][i] = n-1;
      for (int k = i+1 ; k < n ; k += 1)
      {
        resultat[j*n + i][k] = tableau[j][k-1];
      }
    }
    free(tableau[j]);
  }
  free(tableau);
  return resultat;
}
```

Question P2.3 :

La boucle sur le tableau en entrée sera faite une première fois dans une fonction extérieure, qui permet de calculer la taille du tableau de tableaux à construire. C'est utile à deux titres (vu la structure à renvoyer, deux allocations ne sont pas forcément agréables).

Après changement d'avis, la mémoire est libérée dans cette fonction, avec redirection des pointeurs vers le tableau à construire au lieu de recopier le contenu (ce qui rend `n` inutile), et libération des autres pointeurs.

```
int nombre_peen_restantes(int** peen, int taille, int i, int j, bool b)
{
  int nombre_restantes = 0;
  for (int ind = 0 ; ind < taille ; ind += 1)
  {
    if ((peen[ind][i] < peen[ind][j]) == b) nombre_restantes += 1;
  }
  return nombre_restantes;
}

int** filtre_peen(int** peen, int* taille, int i, int j, bool b)
{
  int nombre_restantes = nombre_peen_restantes(peen, *taille, i, j, b);
  int** reponse = malloc(nombre_restantes * sizeof(int*));
  int indice_reponse = 0;
  for (int ind = 0 ; ind < *taille ; ind += 1)
  {
    if ((peen[ind][i] < peen[ind][j]) == b)
    {
      reponse[indice_reponse] = peen[ind];
      indice_reponse += 1;
    }
    else free(peen[ind]);
  }
  *taille = nombre_restantes;
  return reponse;
}
```

Question P2.4 :

C'était une bonne idée de séparer le travail en deux dans la question précédente.

```
bool majorite_filtre_peen(int** peen, int taille, int i, int j)
{
  int nombre_restantes_true = nombre_peen_restantes(peen, taille, i, j, true);
  return nombre_restantes_true > taille - nombre_restantes_true;
}
```

Question P2.5 :

```
void meilleur_filtre_peen(int** peen, int taille, int n, int* pi, int* pj)
{
  int minmax = taille;
  for (int i = 0 ; i < n-1 ; i += 1)
  {
    for (int j = i+1 ; j < n ; j += 1)
    {
      int nombre_restantes = nombre_peen_restantes(peen, taille, i, j, true);
      if (nombre_restantes <= taille / 2) nombre_restantes = taille - nombre_restantes;
      if (nombre_restantes < minmax)
      {
        minmax = nombre_restantes; *pi = i; *pj = j;
      }
    }
  }
}
```

Pas besoin de modification (ni même d'utilisation) de la fonction de la question précédente vu que cela ne coûte pas trop de lignes d'utiliser la fonction extérieure (sans elle, ce serait une autre histoire).

Plutôt que d'allouer de la mémoire pour un couple d'entiers ou de créer une structure correspondante, on a muté deux pointeurs pour une fois.

Question P2.6 :

```
int* tri_minmax(int* tableau, int n)
{
  int** peen = engendre_peen(n);
  int taille = factorielle(n);
  int i; int j;
  while (taille > 1)
  {
    meilleur_filtre_peen(peen, taille, n, &i, &j);
    int** nv_peen = filtre_peen(peen, &taille, i, j, tableau[i] < tableau[j]);
    free(peen);
    peen = nv_peen;
  }
  int* reponse = peen[0];
  free(peen); // Ne pas oublier !
  return reponse;
}
```

Question P2.7 :

La formule de Stirling donne un équivalent en $+\infty$ de $n!$, à savoir $\sqrt{2\pi n}(\frac{n}{e})^n$. En faisant passer le logarithme aux équivalents et en négligeant ce qui est négligeable, on obtient que $\log_2 n! \sim_{n\to+\infty} n\log_2 n$ (en pratique valable pour toutes les bases de logarithme).

Le nombre de comparaisons effectué est en pratique de l'ordre du logarithme en base deux de la factorielle de n, car dans le minmax on peut s'arranger pour que chaque comparaison sépare la liste des peen possibles en deux listes de même taille à un près.

On notera qu'un minmax complet anticiperait les comparaisons suivantes pour être sûr de minimiser la profondeur de l'arbre.

Question P2.8 :

Au vu des dépendances, il faut calculer la complexité de toutes les fonctions écrites. On note n la taille de la liste à trier.

`engendre_peen` : Espace de l'ordre de $n!$ donc temps supérieur ou égal. C'est déjà mal parti !

`filtre_peen` : Temps et espace linéaires en la taille de la liste en entrée (mais qui sera là aussi de l'ordre de $n!$, oups !).

`majorite_filtre_peen` : identique à `filtre_peen`.

`meilleur_filtre_peen` : avec la double boucle, c'est de l'ordre de n^2 fois la complexité des fonctions précédentes, donc on en est à $\mathcal{O}(n^2 n!)$.

`tri_minmax` : D'après la question précédente, le nombre de tours de boucle est de l'ordre de $n\log_2 n$ (en admettant que la meilleure comparaison permette toujours d'avoir deux sous-tableaux équilibrés), ce qui est encore l'occasion de faire une multiplication et mène à $\mathcal{O}(n^3 n!\log_2 n)$.

Cerise sur le gâteau : on peut appliquer la peen finale pour faire un tri qui prend un temps supplémentaire linéaire en la taille du tableau à la fin.

Problème 3

Question P3.1 :

Étape 1 :

```
let sommet1 = { ligne = 0 ; colonne = 1 ; est_tete = false ; taille = 1 }
let sommet2 = { ligne = 2 ; colonne = 4 ; est_tete = true ; taille = 2 }
let sommet3 = { ligne = 3 ; colonne = 2 ; est_tete = false ; taille = 1 }
let sommet4 = { ligne = 4 ; colonne = 0 ; est_tete = false ; taille = 0 }
let sommet5 = { ligne = 4 ; colonne = 4 ; est_tete = false ; taille = 1 }

let graphe1 = { sommets = [ sommet1 ; sommet2 ; sommet3 ; sommet4 ; sommet5 ] ;
arcs = [(sommet2, sommet5)] }
```

Étape 2 :

```
let sommet1 = { ligne = 0 ; colonne = 1 ; est_tete = false ; taille = 1 }
let sommet2 = { ligne = 3 ; colonne = 2 ; est_tete = false ; taille = 1 }
let sommet3 = { ligne = 4 ; colonne = 0 ; est_tete = false ; taille = 0 }
let sommet4 = { ligne = 4 ; colonne = 4 ; est_tete = true ; taille = 3 }

let graphe1 = { sommets = [ sommet1 ; sommet2 ; sommet3 ; sommet4 ] ;
arcs = [(sommet4, sommet2)] }
```

Étape 3 :

```
let sommet1 = { ligne = 0 ; colonne = 1 ; est_tete = false ; taille = 1 }
let sommet2 = { ligne = 3 ; colonne = 2 ; est_tete = true ; taille = 4 }
let sommet3 = { ligne = 4 ; colonne = 0 ; est_tete = false ; taille = 0 }

let graphe1 = { sommets = [ sommet1 ; sommet2 ; sommet3 ] ;
arcs = [(sommet2, sommet1)] }
```

Étape 4 :

```
let sommet1 = { ligne = 0 ; colonne = 1 ; est_tete = true ; taille = 5 }
let sommet2 = { ligne = 4 ; colonne = 0 ; est_tete = false ; taille = 0 }

let graphe1 = { sommets = [ sommet1 ; sommet2 ] ;
arcs = [(sommet1, sommet2)] }
```

Question P3.2 :

```
let matrice_codes =
[|
  [| Vide ; Element 1 ; Vide ; Vide ; Vide |];
  [| Vide ; Vide ; Vide ; Vide ; Vide |];
  [| Vide ; Vide ; Vide ; Tete 1 ; Element 1 |];
  [| Vide ; Vide ; Element 1 ; Vide ; Vide |];
  [| Hautdeforme ; Vide ; Vide ; Vide ; Element 1 |]
|]
```

Question P3.3 :

Les dimensions de la matrice sont considérées comme constantes et donc non récupérées.

Le principe de la fonction ci-après est de chercher s'il existe quoi que ce soit d'autre que `Vide` et `Hautdeforme` dans la matrice.

```
exception Pas_fini

let est_finale matrice =
  try
    for i = 0 to 4 do
      for j = 0 to 4 do
        match matrice.(i).(j) with
        | Tete _ | Element _ -> raise Pas_fini
        | _ -> ()
      done
    done; true
  with _ -> false
```

Question P3.4 :

On reprend (sans la réécrire) la structure de file de l'exercice 10 du TP 7. Ici, une structure de file bornée peut convenir aussi, en mettant par exemple une borne à vingt-cinq éléments (en pratique moins, vu qu'il faut défiler plusieurs fois pour atteindre toutes les cases, et qu'en pratique chaque défilement ne permet d'enfiler qu'au plus trois fois, éventuellement quatre pour le premier défilement). **La structure est là aussi persistante**, donc le parcours en largeur sera récursif.

```
let distance codes i j =
  let reponse = Array.make_matrix 5 5 42 in reponse.(i).(j) <- 0;
  let rec mouline f =
    if not (est_vide_file f) then let (l, c), reste = defiler_file f in
      if (l, c) = (i, j) || codes.(l).(c) = Vide then (* Pas de propagation à travers un élément ! *)
        let fhaut =
          if l > 0 && reponse.(l-1).(c) = 42
          then (reponse.(l-1).(c) <- reponse.(l).(c) + 1; enfiler_file reste (l-1, c))
          else reste
        in let fbas =
          if l < 4 && reponse.(l+1).(c) = 42
          then (reponse.(l+1).(c) <- reponse.(l).(c) + 1; enfiler_file fhaut (l+1, c))
          else fhaut
        in let fgauche =
          if c > 0 && reponse.(l).(c-1) = 42
          then (reponse.(l).(c-1) <- reponse.(l).(c) + 1; enfiler_file fbas (l, c-1))
          else fbas
        in let fdroite =
          if c < 4 && reponse.(l).(c+1) = 42
          then (reponse.(l).(c+1) <- reponse.(l).(c) + 1; enfiler_file fgauche (l, c+1))
          else fgauche
        in mouline fdroite
      else mouline reste
  in mouline (enfiler_file (creer_file ()) (i, j));
  reponse
```

Question P3.5 :

```
let sommets matrice_codes =
  let rec parcours num =
    if num = 25 then []
    else let l = num / 5 in let c = num mod 5 in
    match matrice_codes.(l).(c) with
    | Vide -> parcours (num + 1)
    | Hautdeforme -> { ligne = l ; colonne = c ; est_tete = false ; taille = 0 } :: parcours (num + 1)
    | Tete i -> { ligne = l ; colonne = c ; est_tete = true ; taille = i } :: parcours (num + 1)
    | Element i -> { ligne = l ; colonne = c ; est_tete = false ; taille = i } :: parcours (num + 1)
  in parcours 0
```

```
let rec trouve_sommet l c liste = match liste with
| [] -> failwith "Il fallait trouver quelque chose"
| s::_ when s.ligne = l && s.colonne = c -> s
| _::q -> trouve_sommet l c q

let pas_vide_ni_tete code = match code with
| Vide | Tete _ -> false
| _ -> true

let construit_graphe matrice_codes =
  let som = sommets matrice_codes in
  let construit_arcs s =
    if s.taille = 0 then []
    else let dist = distance matrice_codes s.ligne s.colonne in
      let rec tester num =
        if num = 25 then []
        else let ligne = num / 5 in let colonne = num mod 5 in
          if dist.(ligne).(colonne) = s.taille && pas_vide_ni_tete matrice_codes.(ligne).(colonne) then
            (s, trouve_sommet ligne colonne som)::(tester (num+1))
          else tester (num+1)
      in tester 0
  in let ar = List.fold_left (fun accu s -> construit_arcs s @ accu) [] (List.rev som)
  (* On a utilisé List.rev pour faire joli au niveau de l'ordre des arcs. *)
  in { sommets = som ; arcs = ar }

let reconstruit_codes graphe =
  let reponse = Array.make_matrix 5 5 Vide in
  let place_code s =
    let i = s.ligne in let j = s.colonne in
    if s.est_tete then reponse.(i).(j) <- Tete s.taille
    else if s.taille = 0 then reponse.(i).(j) <- Hautdeforme
    else reponse.(i).(j) <- Element s.taille
  in List.iter place_code graphe.sommets;
  reponse
```

Pour la reconstruction de la matrice de codes, les arcs ne jouent pas le moindre rôle.

Question P3.6 :

On déclenche une erreur tout de même pour les déplacements de nature interdite, pas si le souci concerne la distance. C'est essentiellement afin de rendre le filtrage exhaustif sans prévoir de cas aberrants.

```
let deplacement matrice_codes (ld, cd) (la, ca) =
  let reponse = Array.make_matrix 5 5 Vide in
  for i = 0 to 4 do
    for j = 0 to 4 do
      reponse.(i).(j) <- matrice_codes.(i).(j)
    done
  done;
  begin
    match reponse.(ld).(cd), reponse.(la).(ca) with
    | _, Hautdeforme -> ()
    | Tete n1, Element n2 -> reponse.(la).(ca) <- Tete (n1+n2)
    | Element n1, Element n2 -> reponse.(la).(ca) <- Element (n1+n2)
    | _ -> failwith "Mouvement interdit"
  end;
  reponse.(ld).(cd) <- Vide;
  reponse
```

Question P3.7 :

Il suffit de construire le graphe et de traiter les arcs.

```
let mouvements_possibles matrice_codes =
  let graphe = construit_graphe matrice_codes in
  List.map (fun (s1, s2) -> (s1.ligne, s1.colonne), (s2.ligne, s2.colonne)) graphe.arcs
```

Question P3.8 :

```
let deserialisation chaine =
  let matrice = Array.make_matrix 5 5 Vide in
  for num = 0 to 24 do
    let l = num / 5 in let c = num mod 5 in
    match chaine.[num] with
    | 'H' -> matrice.(l).(c) <- Hautdeforme
    | 'T' -> matrice.(l).(c) <- Tete 1
    | chf when '1' <= chf && chf <= '9' -> matrice.(l).(c) <- Element (int_of_char chf - 48)
    | _ -> ()
  done;
  matrice
```

Question P3.9 :

C'est seulement ici qu'on interdit d'envoyer la tête dans le haut-de-forme alors qu'il reste des éléments ailleurs. Cela revient à bannir tout élément de la pile associé à une matrice non finale sans constructeur `Tete`.

```
exception Tete_trouvee

let perdu_la_tete matrice =
  try
    for i = 0 to 4 do
      for j = 0 to 4 do
        match matrice.(i).(j) with
        | Tete _ -> raise Tete_trouvee
        | _ -> ()
      done
    done; true
  with _ -> false

let resoudre matrice_codes =
  let rec mouline pile = match pile with
    | [] -> failwith "Aucune solution !"
    | (mat, depl)::q -> if est_finale mat then List.rev depl
        else if perdu_la_tete mat then mouline q
        else let possibilites = mouvements_possibles mat in
          let fonction_a_folder pile (dep, arr) = (deplacement mat dep arr, (dep, arr)::depl)::pile in
          mouline (List.fold_left (fonction_a_folder) q (List.rev possibilites))
  in mouline [(matrice_codes, [])]
```

Question P3.10 :

```
let imprime_solution chaine =
  let matrice_codes = deserialisation chaine in
  let solution = resoudre matrice_codes in
  List.iter
  (
    fun ((ld, cd), (la, ca)) -> Printf.printf "Déplacer de (%d, %d) vers (%d, %d).\n" ld cd la ca
  ) solution
```

Question P3.11 :

Flemme de le refaire de tête, donc demandons gentiment au programme écrit tout exprès...

```
Déplacer de (1, 1) vers (1, 2).
Déplacer de (1, 0) vers (0, 1).
Déplacer de (0, 1) vers (3, 1).
Déplacer de (1, 2) vers (0, 0).
Déplacer de (3, 1) vers (1, 3).
Déplacer de (0, 0) vers (1, 3).
```