Correction du DS 6

Julien Reichert

La correction des questions de cours étant notamment dans le cours ou dans les corrigés de TD, elle ne sera pas donnée ici...

Problème 1

Exercice P1.1

```
SELECT T1.Elt AS Elt1, T2.Elt AS Elt2 FROM T AS T1 JOIN T AS T2 WHERE f(T1.Elt,T2.Elt)}
```

Exercice P1.2

```
SELECT COUNT(*) FROM TT WHERE Elt1 = x
```

Exercice P1.3

```
SELECT COUNT(*) FROM TT WHERE Elt1 = Elt2
```

Exercice P1.4

```
SELECT COUNT(*) FROM (SELECT Elt2 AS Elt FROM TT WHERE Elt1 = x
UNION SELECT Elt1 AS Elt FROM TT WHERE Elt2 = x)
```

Problème 2

Exercice P2.1

```
SELECT MIN(Date) FROM Notes
```

Exercice P2.2

```
SELECT COUNT(DISTINCT Classe) FROM Comptes
```

Exercice P2.3

```
SELECT MAX(Note) FROM Notes JOIN Comptes ON Etudiant = Login WHERE Nom = id
```

Exercice P2.4

```
SELECT Etudiant, Date, Titre_DS, Note FROM Notes LEFT JOIN Comptes ON Etudiant = Login
WHERE Login IS NULL
```

Exercice P2.5

```
SELECT Login, COUNT(Note) FROM Comptes LEFT JOIN Notes ON Login = Etudiant GROUP BY Login
```

Exercice P2.6

```
SELECT N1.Titre_DS, N2.Titre_DS
FROM Notes AS N1 JOIN Notes AS N2 ON N1.Date = N2.Date AND N1.Titre_DS < N2.Titre_DS
GROUP BY N1.Titre_DS, N2.Titre_DS
```

Exercice P2.7

```
SELECT Classe, COUNT(*) AS Effectif FROM Comptes GROUP BY Classe ORDER BY COUNT(*) DESC
```

Exercice P2.8

```
SELECT (
    SELECT COUNT(Note) AS Effectif
    FROM Comptes JOIN Notes ON Login = Etudiant
    WHERE Classe = 'PC'
    GROUP BY Titre_DS
    ORDER BY COUNT(*) LIMIT 1
  ) / (SELECT COUNT(*) FROM Comptes WHERE Classe='PC');
```

Le pourcentage évoqué dans l'énoncé ne servait qu'à préciser que l'assiduité était une proportion, mais on acceptera bien évidemment une multiplication par cent du résultat.

On sépare la requête en deux pour éviter une jointure pénible et compliquant les choses (si un étudiant n'est présent à aucun DS et ne rend aucun travail, typiquement).

Problème 3

Exercice P3.1

```
int* suivant(int* tableau, int taille)
{
  int* reponse = malloc(taille * sizeof(int));
  int indice = taille-1;
  while (indice >= 0 && tableau[indice] == 1)
  {
    reponse[indice] = 0;
    indice -= 1;
  }
  if (indice == -1)
  {
    free(reponse);
    return 1;
  }
  reponse[indice] = 1;
  for (int i = 0 ; i < indice ; i += 1) reponse[i] = tableau[i];
  return reponse;
}
```

Exercice P3.2

Plutôt que de faire le calcul dans l'ordre depuis 0, mieux vaut constater que le dernier bit change à chaque fois, l'avant-dernier une fois sur deux, l'antépénultième une fois sur quatre, etc. Donc le nombre de changements de chaque bit correspond à une suite géométrique diminuée d'un (parce qu'on ne fait pas l'étape finale de revenir à zéro) dont la somme est $2^{n+1} - n - 2$ (on a aussi la formule de récurrence $c_n = 2c_{n-1} + n$).

Exercice P3.3

On a successivement 0, 1, 3, 2, 6, 7, 5, 4, 12, 13, 15, 14, 10, 11, 9, 8.

Exercice P3.4

Il est à noter que pour une valeur trop grande de n, le dépassement de capacité menace de toute manière, donc le dépassement arithmétique n'est pas le seul souci.

```
int** gray_recursif(int n)
{
  if (n <= 0 || n > 30) exit(1);
  int nombre_tableaux = 2;
  int** reponse = malloc(2 * sizeof(int*));
  reponse[0] = malloc(sizeof(int));
  reponse[0][0] = 0;
  reponse[1] = malloc(sizeof(int));
  reponse[1][0] = 1;
  for (int iteration = 2 ; iteration <= n ; iteration += 1) // iteration = taille des tableaux
  {
    int** reponsebis = malloc(2 * nombre_tableaux * sizeof(int*));
    for (int i = 0 ; i < nombre_tableaux ; i += 1)
    {
      reponsebis[i] = malloc(iteration * sizeof(int);
      reponsebis[2 * nombre_tableaux - 1 - i] = malloc(iteration * sizeof(int);
      reponsebis[i][0] = 0;
      reponsebis[2 * nombre_tableaux - 1 - i][0] = 1;
      for (int j = 1 ; j < iteration ; j += 1)
      {
        reponsebis[i][j] = reponse[i][j-1];
        reponsebis[2 * nombre_tableaux - 1 - i][j] = reponse[i][j-1];
      }
    }
    for (int i = 0 ; i < nombre_tableaux ; i += 1) free(reponse[i]);
    free(reponse);
    reponse = reponsebis;
    nombre_tableaux *= 2;
  }
  return reponse;
}
```

Exercice P3.5

On montre cette propriété par récurrence sur n, l'initialisation étant directe.

Passer de n à $n+1$ positions selon la méthode donnée fait que les 2^n (nombre de valeurs obtenu par une récurrence évidente) tableaux sont répétés deux fois dans l'ordre puis dans l'ordre inverse et précédés d'un 0 pour la première moitié et d'un 1 pour la deuxième.On a donc tous les codes possibles commençant par un 0 dans la première moitié (hypothèse de récurrence + distinction de cas sur l'ensemble des tableaux de $n+1$ booléens) et tous les codes possibles commençant par un 1 dans la deuxième.

De plus, au vu de l'ordre retenu, au sein de chaque moitié seul un bit diffère d'un code au suivant, et pour passer de la première à la deuxième moitié le 0 devient un 1 mais le reste est égal puisqu'on met en face le dernier code de la première moitié avec le premier de la deuxième, les deux provenant du même code à l'étape précédente.

Le principe de récurrence conclut.

Exercice P3.6

On procède également par récurrence sur n ici. Pour $n = 0$, on constate bien que le dernier tableau contient le code de la valeur 1, et en passant de n à $n + 1$ le dernier tableau est le premier tableau de l'itération précédente (contenant 0, ce qui peut aussi se prouver par une récurrence triviale) avec un 1 devant, donnant bien le nombre 2^{n-1} en binaire.

En pratique ici, la vraie récurrence est celle qui permet de prouver que le premier code est celui de 0, pour le reste on ne se sert pas de l'hypothèse de récurrence.

Exercice P3.7

```
int valuation_dyadique(int i)
{
  assert(i > 0);
  int reponse = 0;
  while (i % 2 == 0)
  {
    reponse += 1;
    i //= 2;
  }
  return reponse;
}
```

Exercice P3.8

On commencera par deux fonctions auxiliaires.

```
int puissance2(int n)
{
  int reponse = 1;
  for (int i = 0 ; i < n ; i += 1) reponse *= 2;
  return reponse;
}

void recopie_tableau(int* destination, int* source, int taille)
{
  for (int i = 0 ; i < taille ; i += 1)
    destination[i] = source[i];
}

int** gray_iteratif(int n)
{
  if (n < 0 || n > 30) exit(1);
  int p2n = puissance2(n);
  int** reponse = malloc(p2n * sizeof(int*));
  reponse[0] = malloc(n * sizeof(int));
  for (int i = 0 ; i < n ; i += 1) reponse[0][i] = 0;
  for (int i = 0 ; i < p2n ; i += 1)
  {
    recopie_tableau(reponse[i], reponse[i-1], n);
    int indice = n - 1 - valuation_dyadique(i);
    reponse[i][indice] = 1 - reponse[i][indice];
  }
  return reponse;
}
```

Exercice P3.9

Il est immédiat que le passage d'un code à l'autre se fait en changeant uniquement un bit, cela se voit dans la boucle remplissant les tableaux.

Concernant l'unicité des valeurs dans le tableau, mieux vaut montrer que les deux méthodes conduisent au même résultat par récurrence sur n.

Initialisation : que c'est vrai pour $n = 1$.

Hérédité : pour passer de n à $n+1$ on a premièrement la bascule du bit à la position 0 qui se fait après 2^n étapes (donc quand la valuation dyadique de l'entier est n, ce qui correspond bien à l'indice attendu pour des tableaux de taille $n+1$).

Deuxièmement, les changements effectués aux $2^n - 1$ premières étapes sont les mêmes qu'à l'étape n, ce qui tombe bien car on veut les mêmes nombres, et pour les $2^n - 1$ dernières étapes, il s'avère que la valuation dyadique de $2^n - i$ est la même que la valuation dyadique de $2^n + i$ (et que celle de i) pour tout i dans l'intervalle (simple factorisation), donc la séquence des $2^n - 1$ valuations dyadiques est la même à gauche et à droite de l'étape 2^n dans le tableau produit par la méthode 1.

L'hérédité est prouvée en faisant intervenir l'hypothèse de récurrence.

Le principe de récurrence conclut.

Exercice P3.10

Dans un algorithme d'exploration exhaustive, passer d'une instance à tester à l'autre peut aller plus vite si les instances sont explorées dans l'ordre du code Gray.

Prenons l'exemple du problème de la sous-somme : plutôt que d'énumérer les 2^n tableaux de booléens correspondant aux sous-listes d'une liste de taille n et de calculer la somme des éléments de la liste pour lesquels le booléen dans le tableau actuel est à vrai en partant de zéro, il est possible d'énumérer les tableaux selon le code Gray et de mettre à jour la somme d'un test à l'autre en additionnant ou en soustrayant la valeur correspondant au seul booléen modifié.

D'autres exemples sont imaginables à partir des considérations mentionnées dans l'énoncé, mais on privilégiera un autre interlocuteur...

Problème 4

Exercice P4.1

Il y a $n+1$ valeurs (de 0 à n) appariées avec toutes les autres valeurs (n occurrences) et avec elles-mêmes (2 cases utilisées à chaque fois), soit un total de $(n+1)(n+2)$.

Autre façon de faire : il y a $\binom{n+1}{2}$ dominos au total et chacun occupe deux cases, pour le même résultat.

Exercice P4.2

On remarque que $(n+1)^2 < (n+1)(n+2) < (n+2)^2$ pour tout entier naturel n.

Attention : on prendra garde à ne pas mélanger entiers et flottants.

```
let n_grille nl nc =
  let n_plus_1_fois_n_plus_2 = float_of_int (nl * nc) in
  int_of_float (sqrt n_plus_1_fois_n_plus_2) - 1;;
```

Exercice P4.3

```
exception Ambigu;;

let placer_double grille k =
  let nl = Array.length grille in
  let nc = Array.length grille.(0) in (* on admet que la grille n'est pas vide *)
  let deja_trouve = ref false in
  let reponse = ref (-1, -1, false) in
  try
    for i = 0 to nl - 1 do
      for j = 0 to nc - 1 do
        if grille.(i).(j) = k then
        begin
          if i < nl - 1 && grille.(i+1).(j) = k then
            if !deja_trouve then raise Ambigu
            else (reponse := (i, j, false); deja_trouve := true);
          if j < nc - 1 && grille.(i).(j+1) = k then
            if !deja_trouve then raise Ambigu
            else (reponse := (i, j, true); deja_trouve := true);
        end
      done
    done;
    !reponse
  with Ambigu -> (-1, -1, false);;
```

La complexité est linéaire en la taille de la grille (cela se voit directement), donc d'après la première question on est en $\mathcal{O}(n^2)$.

Exercice P4.4

La condition initiale `grille.(i).(j) = k` ne peut plus être factorisée et les deux tests internes contiennent une disjonction sur les possibilités de placement des deux valeurs à tester entre la case et la voisine (toujours en cas d'existence).

```
let placer1 grille k1 k2 =
  let nl = Array.length grille in
  let nc = Array.length grille.(0) in (* on admet que la grille n'est pas vide *)
  let deja_trouve = ref false in
  let reponse = ref (-1, -1, false) in
  try
    for i = 0 to nl - 1 do
      for j = 0 to nc - 1 do
        if i < nl - 1 && (grille.(i).(j) = k1 && grille.(i+1).(j) = k2
                       || grille.(i).(j) = k2 && grille.(i+1).(j) = k1) then
          if !deja_trouve then raise Ambigu
          else (reponse := (i, j, false); deja_trouve := true);
        if j < nc - 1 && (grille.(i).(j) = k1 && grille.(i).(j+1) = k2
                       || grille.(i).(j) = k2 && grille.(i).(j+1) = k1) then
          if !deja_trouve then raise Ambigu
          else (reponse := (i, j, true); deja_trouve := true);
      done
    done;
    !reponse
  with Ambigu -> (-1, -1, false);;
```

Exercice P4.5

On remarquera que la fonction `placer_double` est inutile car pour la même complexité asymtotique la fonction `placer1` fait la même chose. La checklist contient les deux positions de chaque domino placé (type option) en prévision des questions suivantes.

```
let premiers_tests grille =
  let nl = Array.length grille in let nc = Array.length grille.(0) in
  let n = n_grille nl nc in
  let progression = Array.make_matrix nl nc false in
  let checklist = Array.make_matrix (n+1) (n+1) None in
  for i = 0 to n do
    for j = 0 to i do
      let (lg, col, sens) = placer1 grille i j in
      if (lg, col, sens) <> (-1, -1, false) then
      ( progression.(lg).(col) <- true;
        if sens then progression.(lg).(col+1) <- true
        else progression.(lg+1).(col) <- true;
        checklist.(i).(j) <- Some ((lg, col), (if sens then (lg, col+1) else (lg+1, col))) )
    done
  done; (progression, checklist);;
```

La complexité est en $\mathcal{O}(n^4)$ pour le nombre quadratique d'appels à la fonction en temps quadratique.

Exercice P4.6

```
(* Fonction à la « table de comptage » *)
let bump_dictionnaire dico cle couple = match Hashtbl.find_opt dico cle with
  | Some m -> Hashtbl.replace dico cle (couple::m)
  | None -> Hashtbl.add dico cle [couple];;

(* Fonction utile pour ne pas avoir besoin de rendre la checklist symétrique *)
let check checklist i j = checklist.(max i j).(min i j);;

let initialise_avancee grille progression checklist =
  let nl = Array.length grille in let nc = Array.length grille.(0) in
  let avancee = Array.make_matrix nl nc (Hashtbl.create 8) in
  for i = 0 to nl-1 do for j = 0 to nc-1 do avancee.(i).(j) <- Hashtbl.create 8 done done;
  (* Les dictionnaires n'étaient pas indépendants ! *)
  for i = 0 to nl - 1 do
    for j = 0 to nc - 1 do
      if not progression.(i).(j) then
      begin
        if i > 0 && not progression.(i-1).(j) && check checklist grille.(i).(j) grille.(i-1).(j) = None
          then bump_dictionnaire avancee.(i).(j) grille.(i-1).(j) (i-1, j);
        if i < nl-1 && not progression.(i+1).(j) && check checklist grille.(i).(j) grille.(i+1).(j) = None
          then bump_dictionnaire avancee.(i).(j) grille.(i+1).(j) (i+1, j);
        if j > 0 && not progression.(i).(j-1) && check checklist grille.(i).(j) grille.(i).(j-1) = None
          then bump_dictionnaire avancee.(i).(j) grille.(i).(j-1) (i, j-1);
        if j < nc-1 && not progression.(i).(j+1) && check checklist grille.(i).(j) grille.(i).(j+1) = None
          then bump_dictionnaire avancee.(i).(j) grille.(i).(j+1) (i, j+1);
        if Hashtbl.length avancee.(i).(j) = 0
          then failwith "Gros souci, résolution impossible à ce stade !" (* vérification facultative *)
      end
    done
  done; avancee;;
```

Exercice P4.7

Règle numéro 1 : Si pour une case donnée, seule une case voisine est libre, on place le domino, qu'on élimine de la checklist et de l'avancée pour toutes les autres cases (et on met à jour la progression pour les deux cases rencontrées).

Règle numéro 1bis : Si pour une case donnée seule une valeur est disponible parmi les voisines mais plus d'une fois, on élimine le domino associé de l'avancée pour toutes les cases autres que les voisines concernées (en pratique pour toutes les cases autres que les voisines, pour un résultat équivalent).

Règle numéro 2 : Si pour un domino donné seules deux cases permettent son placement au vu de la checklist et de l'avancée, on place le domino à l'endroit correspondant. [Cette règle est déjà implémentée dans la fonction `placer1` mais sans utiliser les mêmes arguments.]

Règle numéro 2bis : Sinon, si on a au moins que pour un domino donné toutes les possibilités restantes de placement ont une case en commun, cette case ne peut alors plus avoir dans son dictionnaire correspondant à l'avancée d'autre option que la valeur correspondant au domino à placer.

Les règles 1 et 1bis seront rassemblées en une fonction mais avec des fonctions auxiliaires séparées. Cela permettra de factoriser le code en utilisant quelques variables en plus dans les fonctions auxiliaires.

De même pour les règles 2 et 2bis.

```
(* Booléen indiquant si une seule valeur est disponible dans le voisinage d'une case,
libre si le dictionnaire n'est pas vide dans l'avancée,
puis si oui la valeur disponible puis le nombre de voisines ayant cette valeur. *)
let une_voisine_disponible dico =
  if Hashtbl.length dico <> 1 then (false, 0, [])
  else Hashtbl.fold (fun cle liste accu -> (true, cle, liste)) dico (false, 0, []);;
(* Seul moyen de récupérer l'information sans trop s'embêter : itérer. *)

let rec retirer2 cpl1 cpl2 liste = match liste with
| [] -> []
| a::q when a = cpl1 || a = cpl2 -> retirer2 cpl1 cpl2 q
| a::q -> a::retirer2 cpl1 cpl2 q;;

let retirer_voisin_place dico cpl1 cpl2 cle liste =
  let l2 = retirer2 cpl1 cpl2 liste in
  if l2 = [] then Hashtbl.remove dico cle
  else if l2 <> liste then Hashtbl.replace dico cle l2;;

let place_voisin_restant grille avancee checklist i j v =
  let (i2, j2) = List.hd (Hashtbl.find avancee.(i).(j) v) in
  let vbis = grille.(i).(j) in
  Hashtbl.clear avancee.(i).(j);
  Hashtbl.clear avancee.(i2).(j2);
  checklist.(max v vbis).(min v vbis) <- Some ((i, j), (i2, j2));
  let nl = Array.length grille in let nc = Array.length grille.(0) in
  for ligne = 0 to nl - 1 do
    for colonne = 0 to nc - 1 do
      if grille.(ligne).(colonne) = v then Hashtbl.remove avancee.(ligne).(colonne) vbis;
      if grille.(ligne).(colonne) = vbis then Hashtbl.remove avancee.(ligne).(colonne) v;
      Hashtbl.iter (fun c l -> retirer_voisin_place avancee.(ligne).(colonne) (i, j) (i2, j2) c l)
        avancee.(ligne).(colonne)
    done
  done;;
```

```
let elimine_avancee grille avancee i j v1 v2 =
  let evol = ref false in
  let nl = Array.length grille in let nc = Array.length grille.(0) in
  for i2 = 0 to nl - 1 do
    for j2 = 0 to nc - 1 do
      if not (List.mem (i, j) [(i2, j2); (i2-1, j2); (i2+1, j2); (i2, j2-1); (i2, j2+1)]) then
      (
        if grille.(i2).(j2) = v1 && Hashtbl.mem avancee.(i2).(j2) v2 then
        (evol := true; Hashtbl.remove avancee.(i2).(j2) v2);
        if grille.(i2).(j2) = v2 && Hashtbl.mem avancee.(i2).(j2) v1 then
        (evol := true; Hashtbl.remove avancee.(i2).(j2) v1);
      )
      else if (i2, j2) <> (i, j) && grille.(i2).(j2) = v1 then
        match Hashtbl.find_opt avancee.(i2).(j2) v2 with
        | Some [(i, j)] -> ()
        | None -> ()
        | Some l when List.mem (i, j) l -> Hashtbl.replace avancee.(i2).(j2) v2 [(i, j)]; evol := true
        | Some l -> Hashtbl.remove avancee.(i2).(j2) v2; evol := true
      else if (i2, j2) <> (i, j) && grille.(i2).(j2) = v2 then
        match Hashtbl.find_opt avancee.(i2).(j2) v1 with
        | Some [(i, j)] -> ()
        | None -> ()
        | Some l when List.mem (i, j) l -> Hashtbl.replace avancee.(i2).(j2) v1 [(i, j)]; evol := true
        | Some l -> Hashtbl.remove avancee.(i2).(j2) v1; evol := true
    done
  done; !evol;;

let regle_1 grille avancee checklist =
  let evol = ref false in
  let nl = Array.length grille in let nc = Array.length grille.(0) in
  for i = 0 to nl - 1 do
    for j = 0 to nc - 1 do
      match une_voisine_disponible avancee.(i).(j) with
      | (false, _, _) -> ()
      | (true, v, [_]) -> place_voisin_restant grille avancee checklist i j v; evol := true
      | (true, v, _) -> if elimine_avancee grille avancee i j grille.(i).(j) v
        then evol := true (* peut-être que plus rien ne s'est passé à ce stade *)
    done
  done; !evol;;

let positions_possibles_domino grille avancee v1 v2 =
  let nl = Array.length grille in let nc = Array.length grille.(0) in
  let rec aux i j =
    if i = nl then []
    else if j = nc then aux (i+1) 0
    else if grille.(i).(j) = v1 && Hashtbl.mem avancee.(i).(j) v2
      then ((i, j), Hashtbl.find avancee.(i).(j) v2)::aux i (j+1)
    else if grille.(i).(j) = v2 && Hashtbl.mem avancee.(i).(j) v1
      then ((i, j), Hashtbl.find avancee.(i).(j) v1)::aux i (j+1)
    else aux i (j+1)
  in aux 0 0;;

let taille_deux l = match l with
| [_; _] -> true
| _ -> false;;
```

```
let placer_domino grille avancee checklist v1 v2 liste = match liste with
| [((i, j), _); ((i2, j2), _)] ->
  checklist.(v1).(v2) <- Some ((i, j), (i2, j2));
  Hashtbl.clear avancee.(i).(j);
  Hashtbl.clear avancee.(i2).(j2);
  let nl = Array.length grille in let nc = Array.length grille.(0) in
  for ligne = 0 to nl - 1 do
    for colonne = 0 to nc - 1 do
      Hashtbl.iter (fun c l -> retirer_voisin_place avancee.(ligne).(colonne) (i, j) (i2, j2) c l)
        avancee.(ligne).(colonne)
    done
  done
| _ -> failwith "Cas impossible";;

let rec partout couple liste = match liste with
| [] -> true
| (a, voisins)::q -> (a = couple || voisins = [couple]) && partout couple q;;

let commun l = match l with
| [] -> failwith "Impossible"
| (a, voisins)::_ ->
  List.fold_left (fun accu cpl -> if partout cpl l then cpl else accu) (-1, -1) (a::voisins);;

let garantir_domino avancee (i, j) v1 v2 =
  let evol = Hashtbl.length avancee.(i).(j) <> 1 in
  let f cle _ = if not (List.mem cle [v1; v2]) then Hashtbl.remove avancee.(i).(j) cle in
  Hashtbl.iter f avancee.(i).(j); evol;;

let regle_2 grille avancee checklist =
  let evol = ref false in
  let n = Array.length checklist - 1 in (* plus pratique que n_grille *)
  for v1 = 0 to n do
    for v2 = 0 to v1 do
      if checklist.(v1).(v2) = None then
        let liste = positions_possibles_domino grille avancee v1 v2 in
        if taille_deux liste then ( evol := true; placer_domino grille avancee checklist v1 v2 liste )
        else match commun liste with
        | (-1, -1) -> ()
        | (i, j) -> if garantir_domino avancee (i, j) v1 v2 then evol := true
    done
  done; !evol;;

let toutes_tentatives grille avancee checklist =
  let b1 = regle_1 grille avancee checklist in
  let b2 = regle_2 grille avancee checklist in
  b1 || b2;;
(* ne pas profiter de l'évaluation paresseuse, faire les deux *)
```

Parlons rapidement de la complexité des fonctions, en signalant que toutes les fonctions sur listes et dictionnaires seront considérées comme de complexité constante vu que les tailles seront au plus de quatre dans tous les contextes où on utilisera ces fonctions.

La fonction `une_voisine_disponible` est en temps constant, la fonction `place_voisin_restant` est en $\mathcal{O}(n^2)$, de même que la fonction `elimine_avancee`, ce qui fait que la fonction `regle_1` est en $\mathcal{O}(n^4)$ au vu de la répétition de l'une des deux fonctions précédentes jusqu'à un nombre quadratique de fois.

La fonction `positions_possibles_domino` est en $\mathcal{O}(n^2)$, la fonction `commun` en $\mathcal{O}(n)$ (vu que le nombre de positions possibles d'un domino est lui-même au plus linéaire en n) mais on aurait pu l'arrêter si la taille de la liste était supérieure ou égale à cinq vu que le résultat serait forcément non dans ce cas, ce qui fait que la fonction `regle_2` est aussi en $\mathcal{O}(n^4)$. C'est globalement la même idée de tester toutes les lignes, toutes les colonnes, toutes les valeurs pour une case et toutes les valeurs pour une voisine...

Exercice P4.8

```
let toutes_tentatives_repetees grille avancee checklist =
  while toutes_tentatives grille avancee checklist do () done;; (* C'est beau non ? *)
```

Pour la preuve de terminaison, on se sert du variant qui est le nombre de choses pouvant évoluer dans la fonction `toutes_tentatives` : placer un domino ou diminuer d'au moins un la valeur associée à une clé quelque part dans l'avancée (avec ou sans retrait de la clé). La progression minimale est cette dernière opération, qui peut se produire au plus de l'ordre de $4(n+1)(n+2)$ fois (chaque dictionnaire a une somme des valeurs associées aux clés majorée par le nombre de voisins de la case correspondante, et on ignore le fait que certaines cases aient moins de quatre voisins pour simplifier). La valeur initiale du variant est donc un entier naturel dont on dispose d'une majoration, et soit le variant diminue strictement par l'effet secondaire du test de la boucle conditionnelle, soit le test est faux et on quitte la boucle. La terminaison est donc garantie est la complexité est grossièrement un $\mathcal{O}(n^6)$ (nombre d'appels à la fonction de la question précédente fois sa complexité).

Exercice P4.9

Proposition de méthode : reconstituer la progression (en tant que copie destructible) à partir de l'avancée, imaginer qu'un domino est placé et impacter les cases correspondantes (selon un placement initialement possible) puis regarder la parité de la taille des composantes connexes de `false` dans la progression avec un algorithme de type « pot de peinture », correspondant à un parcours dans un graphe qui ne sera jamais explicité. Si la parité n'est pas respectée, on retire la possibilité de placer le domino en question dans l'avancée (grille et checklist ne sont pas utilisées).

```
let progression_depuis_avancee avancee =
  let nl = Array.length avancee in
  let nc = Array.length avancee.(0) in
  let rep = Array.make_matrix nl nc false in
  for i = 0 to nl - 1 do
    for j = 0 to nc - 1 do
      if Hashtbl.length avancee.(i).(j) = 0
        then rep.(i).(j) <- true
    done
  done; rep;;

let cc_paires progression =
  let nl = Array.length progression in
  let nc = Array.length progression.(0) in
  let rec diffuse i j =
    if i < 0 || i >= nl || j < 0 || j >= nc || progression.(i).(j) then 0
    else
    (
      progression.(i).(j) <- true;
      1 + diffuse (i-1) j + diffuse (i+1) j + diffuse i (j-1) + diffuse i (j+1)
    )
  in let rec aux i j =
    i = nl
    || j = nc && aux (i+1) 0
    || j < nc && (progression.(i).(j) || diffuse i j mod 2 = 0) && aux i (j+1)
  in aux 0 0;; (* Plus joli qu'avec des if, mais certes plus dangereux ! *)
```

```
let placement_impossible avancee (i, j) (i2, j2) =
  let progression = progression_depuis_avancee avancee in
  progression.(i).(j) <- true;
  progression.(i2).(j2) <- true;
  not (cc_paires progression);;

let rec purge_impossibles avancee (i, j) l = match l with
  | [] -> []
  | (i2, j2)::q when placement_impossible avancee (i, j) (i2, j2) -> purge_impossibles avancee (i, j) q
  | cpl::q -> cpl::purge_impossibles avancee (i, j) q;;

let regle_3 avancee =
  let nl = Array.length avancee in
  let nc = Array.length avancee.(0) in
  let evol = ref false in
  for i = 0 to nl - 1 do
    for j = 0 to nc - 1 do
      let maj_dico avancee (i, j) cle l =
      let lbis = purge_impossibles avancee (i, j) l in
      if l <> lbis then
      ( evol := true; Hashtbl.replace avancee.(i).(j) cle lbis )
      in Hashtbl.iter (fun cle l -> maj_dico avancee (i, j) cle l) avancee.(i).(j)
    done
  done; !evol;;
```

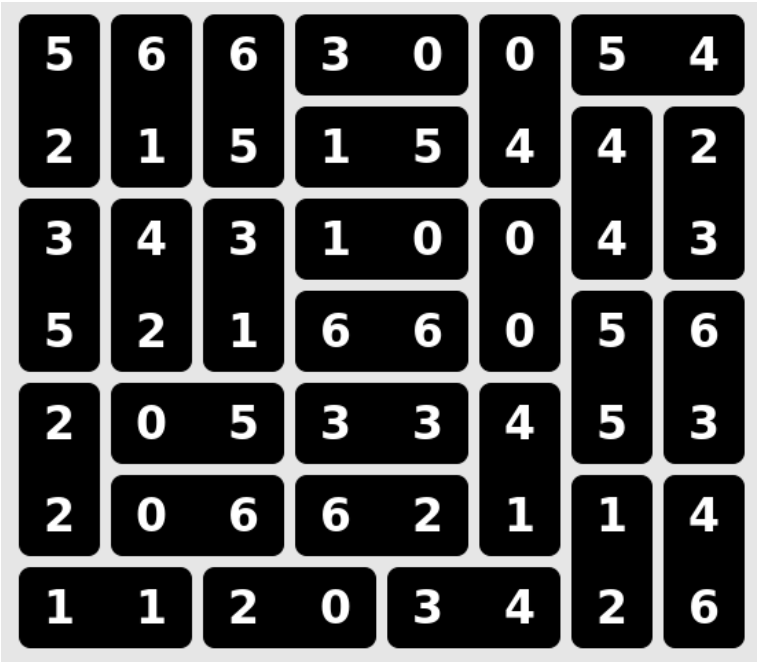
Exercice P4.10

Le plus simple revient à partir sur un backtracking comme pour le sudoku. Une structure de données permet par exemple de recenser les hypothèses faites et, si on accepte d'être très gourmand en espace, de mémoriser l'état des matrices annexes au moment où l'hypothèse a été faite (pour limiter l'espace consommé au prix de complications, demander mon ancien programme de résolution de sudoku en Python ou consulter son adaptation en OCaml potentiellement plus difficile à comprendre dans la correction du TP sur l'exploration exhaustive).

À l'instar du sudoku, le jeu de Dominosa permet cependant d'appliquer des raisonnements avancés, qui sont certes plus difficiles à mettre en œuvre en pratique. Par exemple, si une case où aucun domino n'a encore été placé a une valeur notée k et deux valeurs restantes parmi les voisines et qu'une autre case en-dehors du voisinage a également une valeur notée k et les deux mêmes valeurs restantes parmi les voisines, alors les deux dominos correspondants couvrent forcément ces deux cases et sont à éliminer des cases en-dehors des deux voisinages en question. Le raisonnement est également valable avec trois (voire quatre) possibilités, mais la vérification est plus pénible algorithmiquement et l'efficacité a priori moindre.

Exercice P4.11

Beaucoup de raisonnements des premiers exercices peuvent s'appliquer et mener à une solution plus facilement que dans d'autres instances du jeu que j'ai déjà rencontrées...



Pour être sûr que personne ne termine le DS dans les temps : un sudoku

Écrire le détail du raisonnement serait un massacre écologique si la correction était imprimée, et le temps manque pour tout rédiger à nouveau. Il est envisageable de prendre quelques heures pour le refaire à la main, mais en attendant voici simplement la solution.