Correction du DS 6

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 :

```
let rec toutes_extractions l = match l with
| [] -> []
| a::q -> let couples = toutes_extractions q
  in (a, q) :: List.map (fun (elt, reste) -> (elt, a::reste)) couples

let rec tentative l ltrie =
  let rec mouline extr = match extr, ltrie with
  | [], _ -> []
  | (elt, _) :: q, tete::_ when elt > tete -> mouline q
  | (elt, []) :: _, _ -> elt::ltrie
  | (elt, reste) :: q, _ -> let res = tentative reste (elt::ltrie) in
    if res = [] then mouline q else res
  in mouline (toutes_extractions l)

let tri_backtracking l = tentative l []
```

Exercice 2 :

```
struct trie { char lettre ; bool fini ; struct trie* successeurs ; int nombre_successeurs ; };

typedef struct trie trie;

int qpn(int n)
{
  int res = 1;
  for (int i = 0 ; i < n ; i += 1) res *= 4;
  return res;
}

char correspondances[8][5] = { "ABC" , "DEF" , "GHI" , "JKL" , "MNO" , "PQRS" , "TUV" , "WXYZ" };
// Attention à créer ceci en-dehors de la fonction !

char* correspondance(char c)
{
  if (c < '2' || c > '9') exit(1);
  return correspondances[(int) c - 50];
}
```

```
void recursion(trie dico, char* code, int n, char** res, int* taille_res, char* chaine, int taillechaine)
{
  if (taillechaine == n && dico.fini)
  {
    res[*taille_res] = chaine;
    *taille_res += 1;
    return;
  }
  if (taillechaine == n)
  {
    free(chaine);
    return;
  }
  char* suivants = correspondance(code[taillechaine]);
  for (int i = 0 ; i < dico.nombre_successeurs ; i += 1)
  {
    int indice = 0;
    while (suivants[indice] != '\0')
    {
      if (dico.successeurs[i].lettre == suivants[indice])
      {
        char* nv_chaine = malloc((n+1) * sizeof(char));
        strcpy(nv_chaine, chaine);
        char lettre[2] = { suivants[indice], '\0' };
        strcat(nv_chaine, lettre);
        recursion(dico.successeurs[i], code, n, res, taille_res, nv_chaine, taillechaine+1);
      }
      indice += 1;
    }
  }
  free(chaine);
}

char** mots_possibles(trie dico, char* code, int* taille_res)
{
  int n = strlen(code);
  char** resultat = malloc(qpn(n) * sizeof(char*));
  *taille_res = 0;
  char* chaine = malloc((n+1) * sizeof(char));
  chaine[0] = '\0';
  recursion(dico, code, n, resultat, taille_res, chaine, 0); // n pour ne pas le calculer à chaque fois
  char** vrai_resultat = malloc(*taille_res * sizeof(char*));
  for (int i = 0 ; i < *taille_res ; i += 1)
  {
    vrai_resultat[i] = resultat[i];
  }
  free(resultat);
  return vrai_resultat;
}
```

Exercice 3 :

```
let produit a b = (* ChatGPT par flemme *)
  let n = Array.length a in
  let c = Array.make_matrix n n 0.0 in
  for i = 0 to n - 1 do
    for j = 0 to n - 1 do
      let sum = ref 0.0 in
      for k = 0 to n - 1 do
        sum := !sum +. a.(i).(k) *. b.(k).(j)
      done;
      c.(i).(j) <- !sum
    done
  done;
  c

let transpose m = (* carrée obligatoirement *)
  let n = Array.length m in
  Array.init_matrix n n (fun i j -> m.(j).(i))

let apluslambdab a lam b =
  let n = Array.length a in
  Array.init_matrix n n (fun i j -> a.(i).(j) +. lam *. b.(i).(j))

let carres m =
  let n = Array.length m in
  (
    Array.init_matrix (n/2) (n/2) (fun i j -> m.(i).(j)),
    Array.init_matrix (n/2) (n/2) (fun i j -> m.(i).(j+n/2)),
    Array.init_matrix (n/2) (n/2) (fun i j -> m.(i+n/2).(j)),
    Array.init_matrix (n/2) (n/2) (fun i j -> m.(i+n/2).(j+n/2))
  )

let rec invsdp m =
  let n = Array.length m in
  if n = 1 then [| [| 1. /. m.(0).(0) |] |]
  else let a, tb, b, c = carres m in
  let inva = invsdp a in
  let s = apluslambdab c (-.1.) (produit (produit b inva) tb) in
  let invs = invsdp s in
  let invatbinvs = produit inva (produit tb invs) in
  let hg = apluslambdab inva 1. (produit invatbinvs (produit b inva)) in
  let hd = apluslambdab (Array.make_matrix (n/2) (n/2) 0.) (-.1.) invatbinvs in
  let bg = apluslambdab (Array.make_matrix (n/2) (n/2) 0.) (-.1.) (produit invs (produit b inva)) in
  let bd = invs in
  let reponse = Array.make_matrix n n 0. in
  for i = 0 to n/2 - 1 do
    for j = 0 to n/2 - 1 do
      reponse.(i).(j) <- hg.(i).(j);
      reponse.(i).(j + n/2) <- hd.(i).(j);
      reponse.(i + n/2).(j) <- bg.(i).(j);
      reponse.(i + n/2).(j + n/2) <- bd.(i).(j)
    done
  done;
  reponse
```

```
let invmat m =
  let tmm = produit (transpose m) m in
  produit (invsdp tmm) (transpose m)
```

Exercice 4 :

Soit p_n la complexité du produit matriciel de deux matrices de dimensions (n, n) (p_n peut être négligeable devant n^3, par exemple en utilisant l'algorithme de Strassen). Soit aussi c_n la complexité de l'inversion d'une matrice de dimensions (n, n). Alors l'algorithme donné dans l'énoncé fait un nombre constant de multiplications matricielles et deux inversions de matrices de taille moitié, d'où la formule de récurrence $c_n = 2c_{\frac{n}{2}} + \Theta(p_n)$. Comme p_n est prépondérant devant n^2, le Master Theorem s'applique et donne $c_n = \Theta(p_n)$. Dans la version classique qu'on peut attendre d'un étudiant, $p_n = \Theta(n^3)$ pour utiliser le Master Theorem de l'énoncé, mais sinon une version plus puissante donne le résultat tout de même.

Problème 1

Question P1.1 :

La relation est $1 - *$ car un voyage peut avoir plusieurs étapes, mais une étape n'est reliée qu'à un voyage. Il ne faut pas confondre et penser qu'une destination peut être l'étape de plusieurs voyages : cela correspondrait à des enregistrements distincts de la table `ETAPE` (les dates diffèrent, notamment).

Question P1.2 :

On accepterait à la rigueur `SELECT 'Autriche'`...

```
SELECT DISTINCT pays FROM ETAPE WHERE ville = 'Lermoos'
```

Question P1.3 :

```
SELECT ville FROM ETAPE JOIN VOYAGE ON ETAPE.idvoyage = VOYAGE.idvoyage
WHERE depart = '2025-07-22' ORDER BY arrivee
```

Question P1.4 :

On ne peut pas présumer que la note de dix ait déjà été donnée...

```
SELECT localisation FROM ETAPE
WHERE satisfaction = ( SELECT MAX(satisfaction) FROM ETAPE )
```

Question P1.5 :

```
SELECT transport, COUNT(*) FROM VOYAGE GROUP BY transport
```

Question P1.6 :

```
SELECT SUM(DATEDIFF(day, depart, retour)) FROM VOYAGE
```

Question P1.7 :

```
SELECT ETAPE.* FROM ETAPE JOIN VOYAGE ON ETAPE.idvoyage = VOYAGE.idvoyage
WHERE arrivee < depart OR arrivee > retour
```

Question P1.8 :

```
SELECT VOYAGE.* FROM VOYAGE JOIN ETAPE ON VOYAGE.idvoyage = ETAPE.idvoyage
GROUP BY VOYAGE.idvoyage, VOYAGE.transport, VOYAGE.depart, VOYAGE.retour
HAVING VOYAGE.depart <> MIN(ETAPE.arrivee)
```

Question P1.9 :

```
SELECT COUNT(DISTINCT pays) AS maxpays FROM ETAPE GROUP BY idvoyage ORDER BY maxpays DESC LIMIT 1
```

Question P1.10 :

Avant de se lancer, il faut déjà voir comment récupérer le nombre de jours passés sur une étape. L'idée est de faire une jointure avec une table principale (une copie de la table `ETAPES`) et une table secondaire (une autre copie avec le seul enregistrement directement supérieur en termes de date à l'enregistrement de la première copie). Pour trouver ce seul enregistrement directement supérieur, on recherche une valeur intermédiaire et s'il n'y en a pas c'est gagné. Pour le savoir... on cherche un `NULL` dans une jointure à gauche ! On n'oubliera pas l'étape finale, qui s'obtient selon un principe similaire.

```
SELECT SUM(prix) FROM
(
  SELECT DATEDIFF(E1.arrivee, E2.arrivee) * E1.ppj AS prix
  FROM ETAPE AS E1 JOIN ETAPE AS E2
  ON E1.idvoyage = E2.idvoyage AND E1.arrivee < E2.arrivee
  LEFT JOIN ETAPE AS E3
  ON E2.idvoyage = E3.idvoyage AND E1.arrivee < E3.arrivee AND E3.arrivee < E2.arrivee
  WHERE E1.idvoyage = 1 AND E3.arrivee IS NULL

  UNION

  SELECT DATEDIFF(E1.arrivee, retour) * E1.ppj AS prix
  FROM ETAPE AS E1 JOIN VOYAGE
  ON E1.idvoyage = VOYAGE.idvoyage
  LEFT JOIN ETAPE AS E2
  ON E1.idvoyage = E2.idvoyage AND E1.arrivee < E2.arrivee
  WHERE E1.idvoyage = 1 AND E2.arrivee IS NULL
)
```

Problème 2

Question P2.1 :

Le tableau peut être en-dehors de toute fonction, mais les instructions sur le pointeur de pointeurs et sur les pointeurs en question sont à mettre dans une fonction.

```
int exemple_droite[12][4] =
{
  { 0 , 1 , 2 , 3 },
  { 2 , 4 , 5 , 4 },
  { 5 , 6 , 3 , 7 },
  { 2 , 7 , 1 , 0 },
  { 8 , 1 , 9 , 9 },
  { 7 , 6 , 4 , 4 },
  { 8 , 1 , 7 , 5 },
  { 8 , 3 , 2 , 8 },
  { 6 , 5 , 0 , 6 },
  { 3 , 9 , 0 , 9 },
  { -1 , -1 , -1 , -1 }, // un -1 aurait certes suffi, mais c'est mieux pour le débug
  { -1 , -1 , -1 , -1 },
};

int** ptr_ex_dr = malloc(12 * sizeof(int*));
for (int i = 0 ; i < 12 ; i += 1)
{
  ptr_ex_dr[i] = malloc(4 * sizeof(int));
  for (int j = 0 ; j < 4 ; j += 1)
  {
    ptr_ex_dr[i][j] = exemple_droite[i][j];
  }
}
```

Question P2.2 :

```
bool fini(int** plateau, int nbtubes)
{
  for (int i = 0 ; i < nbtubes ; i += 1)
  {
    if (plateau[i][0] == -1) continue;
    for (int j = 1 ; j < 4 ; j += 1)
    {
      if (plateau[i][j] != plateau[i][0]) return false;
    }
  }
  return true;
}
```

Question P2.3 :

```
int** copie_plateau(int** plateau, int nbtubes)
{
  int** reponse = malloc(nbtubes * sizeof(int*));
  for (int i = 0 ; i < nbtubes ; i += 1)
  {
    reponse[i] = malloc(4 * sizeof(int));
    for (int j = 0 ; j < 4 ; j += 1) reponse[i][j] = plateau[i][j];
  }
  return reponse;
}
```

Question P2.4 :

Avec fonction auxiliaire.

```
int nombre_possibles(int** plateau, int depart, int arrivee)
{
  int couldep = -1;
  int nbdep = 1;
  int coularr = -1;
  int placearr = 0;
  bool depuisfond = true;
  for (int j = 0 ; j < 4 ; j += 1)
  {
    if (plateau[depart][j] == -1) break;
    if (couldep == plateau[depart][j]) nbdep += 1;
    else
    {
      if (couldep != -1) depuisfond = false;
      couldep = plateau[depart][j];
      nbdep = 1;
    }
  }
  for (int j = 0 ; j < 4 ; j += 1)
  {
    if (plateau[arrivee][j] == -1)
    {
      placearr = 4-j;
      break;
    }
    coularr = plateau[arrivee][j];
  }
  if (couldep == -1 || placearr == 0 || (couldep != coularr && coularr != -1)
    || (depuisfond && placearr == 4)) return 0;
  if (nbdep <= placearr) return nbdep;
  else return -placearr - nbdep*5; // Valeur négative pour préparer la gestion de boucles infinies
}
```

```
int* coups_possibles(int** plateau, int nbtubes)
{
  int* reponse = malloc((6 * 5 * nbtubes + 1) * sizeof(int)); // Majoration brutale
  reponse[0] = 0;
  for (int i = 0 ; i < nbtubes ; i += 1)
  {
    for (int j = 0 ; j < nbtubes ; j += 1)
    {
      if (i != j)
      {
        int nb = nombre_possibles(plateau, i, j);
        if (nb > 0)
        {
          reponse[reponse[0] + 1] = i;
          reponse[reponse[0] + 2] = nb;
          reponse[reponse[0] + 3] = j;
          reponse[0] += 3;
        }
        if (nb < 0) // Placer toutes les balles du tube i dans deux endroits
        {
          for (int k = 0 ; k < nbtubes ; k += 1)
          {
            if (i != k && j != k)
            {
              int nb2 = nombre_possibles(plateau, i, k);
              if (nb2 != 0) // < 0 éventuellement
              {
                int nbdep = (-nb) / 5; // décodage
                int vrainbneg = nb + 5 * nbdep;
                reponse[reponse[0] + 1] = i;
                reponse[reponse[0] + 2] = vrainbneg;
                reponse[reponse[0] + 3] = j;
                reponse[reponse[0] + 4] = i;
                reponse[reponse[0] + 5] = nbdep + vrainbneg; // forcément
                reponse[reponse[0] + 6] = k;
                reponse[0] += 6;
              }
            }
          }
        }
      }
    }
  }
  int* vraie_reponse = malloc((reponse[0] + 1) * sizeof(int));
  for (int i = 0 ; i <= reponse[0] ; i += 1) vraie_reponse[i] = reponse[i];
  free(reponse);
  return vraie_reponse;
}

bool perdu(int** plateau, int nbtubes)
{
  int* cp = coups_possibles(plateau, nbtubes);
  bool reponse = cp[0] == 0;
  free(cp); // Ne pas oublier !
  return reponse; // Ou tester d'abord !fini mais pas besoin vu l'ordre d'appel des fonctions.
}
```

Question P2.5 :

```
void imprime_coup(int depart, int nbballes, int arrivee)
{
  if (nbballes < 0) nbballes = -nbballes;
  char pluriel[2] = "s";
  if (nbballes < 2) pluriel[0] = '\0';
  printf("Déplacer %d balle%s de la colonne %d à la colonne %d.\n", nbballes, pluriel, depart, arrivee);
}
```

Question P2.6 :

```
void libere_plateau(int** plateau, int nbtubes)
{
  for (int i = 0 ; i < nbtubes ; i += 1)
  {
    free(plateau[i]);
  }
  free(plateau);
}

void applique_coup(int** plateau, int depart, int nbballes, int arrivee)
{
  int couldep = -1;
  for (int j = 0 ; j < 4 ; j += 1)
  {
    if (plateau[depart][j] == -1)
    {
      for (int k = 1 ; k <= nbballes ; k += 1)
      {
        plateau[depart][j-k] = -1;
      }
      break;
    }
    couldep = plateau[depart][j];
    if (j == 3)
    {
      for (int k = 1 ; k <= nbballes ; k += 1)
      {
        plateau[depart][4-k] = -1;
      }
    }
  }
  for (int j = 0 ; j < 4 ; j += 1)
  {
    if (plateau[arrivee][j] == -1)
    {
      for (int k = 0 ; k < nbballes ; k += 1)
      {
        plateau[arrivee][j+k] = couldep;
      }
      if (j + nbballes < 4) plateau[arrivee][j + nbballes] = -1;
      break;
    }
  }
}
```

```
int* resout(int** plateau, int nbtubes)
{
  if (fini(plateau, nbtubes))
  {
    int* reponse = malloc(sizeof(int));
    reponse[0] = 0;
    return reponse;
  }
  if (perdu(plateau, nbtubes)) return NULL;
  int* cp = coups_possibles(plateau, nbtubes);
  for (int i = 1 ; i < cp[0] ; i += 3)
  {
    int depart = cp[i];
    int nbballes = cp[i+1];
    int arrivee = cp[i+2];
    int** copie = copie_plateau(plateau, nbtubes);
    if (nbballes > 0)
    {
      applique_coup(copie, depart, nbballes, arrivee);
    }
    else
    {
      applique_coup(copie, depart, -nbballes, arrivee);
      applique_coup(copie, cp[i+3], cp[i+4], cp[i+5]);
      i += 3;
    }
    int* resultat = resout(copie, nbtubes);
    if (resultat == NULL)
    {
      libere_plateau(copie, nbtubes);
    }
    else
    {
      int quatre_ou_sept = nbballes > 0 ? 4 : 7;
      int* reponse = malloc((resultat[0] * 3 + quatre_ou_sept) * sizeof(int));
      reponse[0] = resultat[0] + quatre_ou_sept / 3;
      reponse[1] = depart;
      reponse[2] = nbballes > 0 ? nbballes : -nbballes;
      reponse[3] = arrivee;
      if (nbballes < 0)
      {
        reponse[4] = cp[i];
        reponse[5] = cp[i+1];
        reponse[6] = cp[i+2];
      }
      for (int i = 1 ; i <= resultat[0] * 3 ; i += 1) reponse[i+quatre_ou_sept-1] = resultat[i];
      free(resultat);
      libere_plateau(copie, nbtubes);
      free(cp);
      return reponse;
    }
  }
  free(cp);
  return NULL;
}
```

```
void imprime_solution(int* solution)
{
  for (int i = 0 ; i < solution[0] ; i += 1)
  {
    imprime_coup(solution[3*i+1], solution[3*i+2], solution[3*i+3]);
  }
}
```

Problème 3

Question P3.1 :

$$(\neg a \wedge b \wedge c) \vee (a \wedge \neg b \wedge c) \vee (a \wedge b \wedge \neg c)$$

Question P3.2 :

```
let abc =
[|
  [| ("a", false) ; ("b", true) ; ("c", true) |] ;
  [| ("a", true) ; ("b", false) ; ("c", true) |] ;
  [| ("a", true) ; ("b", true) ; ("c", false) |]
|]
```

Question P3.3 :

La réponse est $n - 1$.

Un argument à base de degrés de liberté s'imagine, ou un autre qui fait la part belle à l'algèbre linéaire : on peut imaginer qu'une formule 2T1L concernant trois variables a, b et c équivaut à une équation $a+b+c = 2$ en assimilant le vrai à 1 et le faux à 0, dans ce cas on se place dans un espace de dimension n dont les coordonnées s'expriment en fonction de la valeur de vérité de chaque variable, et $a + b + c = 2$ est l'équation d'un hyperplan affine.

Il vient que l'intersection de $n - 2$ hyperplans affines, si elle n'est pas vide, contient au moins une droite.

Ceci permet de voir qu'avec strictement moins de formules 2T1L, il reste forcément au moins deux interprétations possibles.

Cependant, en fonction des formules données, $n - 1$ formules ne suffisent pas forcément...

Question P3.4 :

```
let variables tab =
  let dico = Hashtbl.create 3 in
  let rec mouline i =
    if i < Array.length tab then
    (
      Hashtbl.add dico (fst tab.(i)) 0;
      mouline (i+1)
    )
  in mouline 0;
  dico
```

```
exception Non

let est_2T1L tab =
  try
    Array.length tab = 3 &&
    let dico = variables tab.(0) in
    for i = 0 to 2 do
      if Array.length tab.(i) <> 3 then raise Non;
      let nbfaux = ref 0 in
      for j = 0 to 2 do
        let (v, b) = tab.(i).(j) in
        match Hashtbl.find_opt dico v with
        | None -> raise Non
        | Some n ->
          if not b then ( incr nbfaux; Hashtbl.replace dico v (n+1) )
      done; if !nbfaux <> 1 then raise Non
    done;
    Hashtbl.iter (fun _ nb -> if nb <> 1 then raise Non) dico;
    true
  with Non -> false
```

Question P3.5 :

En pratique on pouvait déjà en avoir besoin dans la fonction précédente... Et tant qu'à faire, vu ce qui suit, on découpe le travail en deux fonctions.

```
let liste_cles dico =
  Hashtbl.fold (fun nom _ accu -> nom :: accu) dico []

let liste_variables tab =
  let dico = variables tab.(0) in
  liste_cles dico
```

Question P3.6 :

Pour une exploration exhaustive, on va construire toutes les interprétations possibles en tant que dictionnaires (structure d'ensemble en pratique, mais on reste dans le cadre du programme). On aurait pu faire le même travail avec des listes, mais on optimise ainsi le coût de la recherche.

On ne vérifiera pas que la liste ne contient que des formules 2T1L, et le fait que c'en sont rend l'évaluation des formules bien plus simples.

```
let sat2T1L interp tab =
  let l = liste_variables tab in
  List.fold_left (fun accu vari -> accu + if Hashtbl.mem interp vari then 1 else 0) 0 l = 2

let rec satisfait interp tablist = match tablist with
| [] -> true
| tab::q -> sat2T1L interp tab && satisfait interp q

let variables_tablist tablist =
  let d = Hashtbl.create 8 in
  let rec ajoute_var tab =
    for i = 0 to 2 do Hashtbl.replace d (fst tab.(0).(i)) 42 done
  in List.iter ajoute_var tablist;
  d
```

```
let ajoute_partout cle dicos =
  List.map (fun d -> let dbis = Hashtbl.copy d in Hashtbl.add dbis cle 42; dbis) dicos

let tous_sous_dicos dico =
  Hashtbl.fold (fun cle _ accu -> ajoute_partout cle accu @ accu) dico [Hashtbl.create 8]

let satisfaire tablist =
  let dico = variables_tablist tablist in
  let interpretations = tous_sous_dicos dico in
  let rec trouve_interp l = match l with
  | [] -> failwith "Insatisfaisable"
  | interp::_ when satisfait interp tablist -> liste_cles interp
  | _::q -> trouve_interp q
  in trouve_interp interpretations
```

Question P3.7 :

Il suffit de changer la dernière fonction, en reprenant le reste.

Une solution revient à avoir un argument supplémentaire dans la récursion, sous la forme d'une « option de solution ».

```
let satisfaire_unique tablist =
  let dico = variables_tablist tablist in
  let interpretations = tous_sous_dicos dico in
  let rec trouve_interp l dejatrouve = match l, dejatrouve with
  | [], None -> failwith "Insatisfaisable"
  | [], Some interp -> liste_cles interp
  | interp::q, None when satisfait interp tablist -> trouve_interp q (Some interp)
  | interp::_, _ when satisfait interp tablist -> failwith "Au moins deux solutions"
  | _::q, _ -> trouve_interp q dejatrouve
  in trouve_interp interpretations None
```