Correction du DS 3

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 :

Version avec un seul parcours du tableau (plus compliquée mais plus jolie) :

```
int classementsansmajor(int* tab, int taille, int x)
{
  bool present = false;
  int max = tab[0];
  int superieurs = 0;
  int occmax = 1;
  for (int i = 0 ; i < taille ; i += 1)
  {
    if (tab[i] == x) present = true;
    else if (tab[i] > x) superieurs += 1;
    if (tab[i] > max)
    {
      max = tab[i];
      occmax = 1;
    }
    else if (tab[i] == max) occmax += 1;
  }
  if (!present || x == max) exit(1);
  return superieurs - occmax + 1;
}
```

Exercice 2 :

```
let rec quigagne vict f1 f2 = match vict with
| [] -> if f1 = 0 && f2 = 0 then failwith "Personne n'avait d'argent"
        else if f1 = 0 then -1 else if f2 = 0 then 1 else 0
| true::q -> if f1 = 0 || f2 = 0 then failwith "Un joueur est ruiné prématurément";
        if f1 < f2 then quigagne q (2*f1) (f2-f1) else quigagne q (f1+f2) 0
| false::q -> if f1 = 0 || f2 = 0 then failwith "Un joueur est ruiné prématurément";
        if f1 > f2 then quigagne q (f1-f2) (2*f2) else quigagne q 0 (f1+f2)
```

Exercice 3 :

Inutile de faire une allocation mémoire pour quatre valeurs, mais ce n'est pas une erreur.

L'initialisation aurait été plus pénible sans la garantie de positivité des valeurs.

```
int quatrieme_plus_grand(int* tab, int taille)
{
  if (taille < 4) exit(1);
  int meilleurs[4] = { -1 , -1 , -1 , -1 };
  for (int i = 0 ; i < taille ; i += 1)
  {
    int x = tab[i];
    int buff;
    for (int j = 0 ; j < 4 ; j += 1)
    {
      if (x > meilleurs[j])
      {
        buff = meilleurs[j];
        meilleurs[j] = x;
        x = buff;
      }
    }
  }
  return meilleurs[3];
}
```

Invariant de la boucle extérieure : `meilleurs` contient les quatre plus grands éléments vus jusqu'à présent (si on en a vu moins, il s'agit de tous les éléments) dans l'ordre décroissant.

L'hérédité de la boucle extérieure nécessite d'avoir une vue précise sur le fonctionnement de la boucle intérieure. Celle-ci pourrait aussi se prouver à l'aide d'un invariant de boucle, mais le fait que le nombre de tours soit fixé permet de remplacer la preuve usuelle sur les boucles en un « déroulement » de l'ensemble des opérations effectuées.

Exercice 4 :

```
let tous_miroirs tab =
  let mots = Hashtbl.create 8 in
  let rec parcours miroirs indice =
    if indice = Array.length mots then miroirs
    else
      let mot = tab.(indice) in (* pas besoin de begin/end grâce à cela *)
      let taillemot = String.length mot in
      if not (Hashtbl.mem mots mot) then
        let motalenvers = String.init taillemot (fun i -> mot.(taillemot-1-i)) in
        Hashtbl.add mots tab.(indice) 0;
        if motalenvers <> mot && Hashtbl.mem mots motalenvers then
          parcours ((min mot motalenvers, max mot motalenvers)::miroirs) (indice+1)
          else parcours miroirs (indice+1)
      else parcours miroirs (indice+1)
  in parcours [] 0;;
```

La fonction `parcours` termine trivialement (variant pour la récursion : `Array.length tab - indice`). Le nombre d'appels récursifs total est le nombre de mots. Le contenu de chaque appel récursif est un ajout de clé à un dictionnaire sous réserve de non appartenance. Cette opération est de complexité linéaire dans le pire des cas, mais on peut accepter de considérer la complexité comme constante aussi. Le reste est de la manipulation de variables ou de listes en temps constant, mais il y a aussi la récupération du miroir d'une chaîne de caractères et des comparaisons entre chaînes. Ceci n'est certes pas à considérer comme en temps constant, mais en cumulant le tout on se rend compte qu'on reste linéaire en la taille en mémoire de l'entrée (qui est globalement la somme des tailles des chaînes du tableau).

Problème 1

Question P1.1 :

```
struct t_r_i { int taille; int capacite; int* donnees; };
typedef struct t_r_i itr;
itr creer_itr(int capacite);
int taille_itr(itr t);
int acces_itr(itr t, int i);
void modif_itr(itr t, int i, int x);
void append_itr(itr* t, int x);
int pop_itr(itr* t);
```

Question P1.2 :

Un multiensemble sera de type `itr`, d'où les prototypes à venir.

```
itr creer_multiset(); // Capacité arbitraire.
void ajouter_multiset(itr* multiset, int element);
void retirer_multiset(itr* multiset, int element);
bool vide_multiset(itr multiset);
int nombre_occ_multiset(itr multiset, int element);
bool egalite_multiset(itr multiset1, itr multiset2);
bool inclusion_multiset(itr multiset1, itr multiset2);
itr reunion_multiset(itr multiset1, itr multiset2);
itr intersection_multiset(itr multiset1, itr multiset2);
itr somme_multiset(itr multiset1, itr multiset2);
itr difference_multiset(itr multiset1, itr multiset2);
```

Question P1.3 :

Si on veut économiser de la complexité au niveau des accès en écriture dans le tableau, au lieu de faire de simples décalages, on explore depuis le début et on échange uniquement une occurrence de chaque élément différent. Si...

```
void ajouter_multiset(itr* multiset, int element)
{
  int indice = 0;
  while (indice < taille_itr(multiset) && acces_itr(multiset, indice) != element) // trouver
  {
    indice += 1;
  }
  while (indice < taille_itr(multiset) && acces_itr(multiset, indice) == element) // aller au bout
  {
    indice += 1;
  }
  int swap = element;
  for (int i = indice ; i < taille_itr(multiset) ; i += 1)
  {
    int buff = acces_itr(multiset, i);
    modif_itr(multiset, i, swap);
    swap = buff;
  }
  append_itr(&multiset, swap);
}
```

La dernière partie procède aux décalages proprement dits.

Question P1.4 :

Le squelette de la fonction précédente se reporte. En d'autres termes, il aurait été pertinent d'écrire deux fonctions supplémentaires donnant le premier et le dernier indice où se situe un élément d'un multi-ensemble.

```
int nombre_occ_multiset(itr multiset, int element)
{
  while (indice < taille_itr(multiset) && acces_itr(multiset, indice) != element) // trouver
  {
    indice += 1;
  }
  int indice2 = indice;
  while (indice2 < taille_itr(multiset) && acces_itr(multiset, indice2) == element) // aller au bout
  {
    indice2 += 1;
  }
  return indice2-indice; // 0 si l'élément n'apparaît pas
}
```

Question P1.5 :

```
bool inclusion_multiset(itr multiset1, itr multiset2)
{
  int etape = 0;
  while (etape < taille_itr(multiset1))
  {
    int element = acces_itr(multiset1, etape);
    int nb1 = nombre_occ_multiset(multiset1, element);
    int nb2 = nombre_occ_multiset(multiset2, element);
    if (nb1 > nb2) return false;
    int indice = etape;
    while (indice < taille_itr(multiset1) && acces_itr(multiset1, indice) == acces_itr(multiset1, etape)) /
    {
      indice += 1;
    }
    etape = indice;
    // En pratique, on peut fusionner les variables, mais la ligne du while serait trop courte.
  }
  return true;
}

bool egalite_multiset(itr multiset1, itr multiset2)
{
  return inclusion_multiset(multiset1, multiset2) && inclusion_multiset(multiset2, multiset1);
}
```

Question P1.6 :

On va faire un vilain copier-coller pour ne pas trop se fatiguer.

L'idée est d'incorporer dans la réunion tous les éléments du premier qui y apparaissent au moins aussi souvent que dans le deuxième, puis tous les éléments du deuxième qui y apparaissent strictement plus souvent que dans le premier.

Ainsi, rien n'est oublié (si des éléments ne sont que dans l'un des deux) et on n'ajoute pas les éléments deux fois.

```
itr reunion_multiset(itr multiset1, itr multiset2)
{
  int etape = 0;
  itr reunion = creer_multiset();
  while (etape < taille_itr(multiset1))
  {
    int element = acces_itr(multiset1, etape);
    int nb1 = nombre_occ_multiset(multiset1, element);
    int nb2 = nombre_occ_multiset(multiset2, element);
    if (nb1 >= nb2)
    {
      for (int i = 0 ; i < nb1 ; i += 1)
      {
        ajouter_multiset(&reunion, element);
      }
    }
    int indice = etape;
    while (indice < taille_itr(multiset1) && acces_itr(multiset1, indice) == acces_itr(multiset1, etape))
    {
      indice += 1;
    }
    etape = indice;
  }
  etape = 0;
  while (etape < taille_itr(multiset2))
  {
    int element = acces_itr(multiset2, etape);
    int nb1 = nombre_occ_multiset(multiset1, element);
    int nb2 = nombre_occ_multiset(multiset2, element);
    if (nb1 < nb2)
    {
      for (int i = 0 ; i < nb2 ; i += 1)
      {
        ajouter_multiset(&reunion, element);
      }
    }
    int indice = etape;
    while (indice < taille_itr(multiset2) && acces_itr(multiset1, indice) == acces_itr(multiset1, etape))
    {
      indice += 1;
    }
    etape = indice;
  }
  return reunion;
}
```

Pour l'intersection, une seule des deux parties est nécessaire, et on remplace le test conditionnel et son contenu par ceci :

```
if (nb1 > nb2) nb1 = nb2;
for (int i = 0 ; i < nb1 ; i += 1)
{
  ajouter_multiset(&intersection, element); // résultat renommé pour l'occasion
}
```

On notera que `nb1` ne peut pas être nul, mais `nb2` si, et dans ce cas, la boucle est vide, ce qui est cohérent.

Question P1.7 :

Même si ce n'est pas précisé dans la consigne (il faut séparer les questions pour que la correction et la gestion du barème soient facilitées), on peut aussi recycler le code précédent. Il est également autorisé de faire une simple fusion car la fonction d'ajout a été écrite pour respecter la contrainte de rassemblement des valeurs identiques. On notera que coller les tableaux redimensionnables directement sans utiliser `ajouter_multiset` ne serait alors pas correct.

```
itr somme_multiset(itr multiset1, itr multiset2)
{
  itr somme = creer_multiset();
  for (int i = 0 ; i < taille_itr(multiset1) ; i += 1)
  {
    ajouter_multiset(&somme, acces_itr(multiset1, i));
  }
  for (int i = 0 ; i < taille_itr(multiset2) ; i += 1)
  {
    ajouter_multiset(&somme, acces_itr(multiset2, i));
  }
  return somme;
}
```

Question P1.8 :

On reprend le principe de l'intersection ici.

```
itr difference_multiset(itr multiset1, itr multiset2)
{
  int etape = 0;
  itr difference = creer_multiset();
  while (etape < taille_itr(multiset1))
  {
    int element = acces_itr(multiset1, etape);
    int nb1 = nombre_occ_multiset(multiset1, element);
    int nb2 = nombre_occ_multiset(multiset2, element);
    if (nb1 > nb2)
    {
      for (int i = 0 ; i < nb1-nb2 ; i += 1)
      {
        ajouter_multiset(&difference, element);
      }
    }
    int indice = etape;
    while (indice < taille_itr(multiset1) && acces_itr(multiset1, indice) == acces_itr(multiset1, etape))
    {
      indice += 1;
    }
    etape = indice;
  }
  return difference;
}
```

On notera que cette correction propose une implémentation d'efficacité limitée notamment car beaucoup de choses dépendent de la structure de tableau redimensionnable sur laquelle on s'appuie. Changer la structure nécessite de réécrire presque l'intégralité des fonctions de l'interface sans profiter d'astuces qui auraient permis de factoriser le code, par exemple en ajoutant des fonctions communes.

Problème 2

Question P2.1 :

```
let coher c = match c with
| Atout n -> 1 <= n && n <= 22
| _ -> true

let coherente main =
  Array.length main = 9 && Array.for_all coher main
```

Question P2.2 :

```
exception Perdu

let uniques distribution =
  let toutes_cartes = Hashtbl.create 42 in
  try
    for i = 0 to 4 do
      for j = 0 to (if i = 4 then 6 else 9) do
        if Hashtbl.mem toutes_cartes distribution.(i).(j) then raise Perdu;
        Hashtbl.add toutes_cartes distribution.(i).(j)  0
      done
    done;
    true
  with Perdu -> false

let coherente_donne distribution =
  coherente distribution.(0) &&
  coherente distribution.(1) &&
  coherente distribution.(2) &&
  coherente distribution.(3) &&
  Array.length distribution.(4) = 6 &&
  Array.for_all coher distribution.(4) &&
  uniques distribution
```

Question P2.3 :

Une fois de plus dans ce devoir, la complexité dépend de ce que l'on considère comme complexité pour la manipulation de dictionnaires. Ce sera donc linéaire ou quadratique suivant le cas pour la fonction ci-avant.

Ceci étant, on peut improviser soi-même une fonction de hachage et remplacer le dictionnaire par un tableau de booléens de taille quarante-deux, donc cela garantit une complexité linéaire.

Et pour tout dire, vu que l'on va forcément travailler sur un tableau de taille connue (ou alors on détecte en temps constant un problème), le calcul de complexité n'a pas de pertinence et on peut accepter la réponse $\mathcal{O}(1)$.

Question P2.4 :

```
let est_bout c = match c with
| Atout(1) | Atout(21) | Atout(22) -> true
| _ -> false

let contient_bout = Array.exists est_bout
```

Observer l'absence de l'argument `main`, c'est la magie de la curryfication !

Question P2.5 :

La valeur de retour est un tableau de taille quatre, de sorte qu'à l'indice `i` figure le numéro du joueur qui doit faire l'enchère finale `i` afin que le joueur précisé en argument puisse prendre la carte d'indice en argument.

```
let comment_prendre indcarte indjoueur =
  let reponse_0 = (indjoueur - indcarte/2 + 3) mod 4 in
  let reponse_1 = (indjoueur - (indcarte+1)/2 + 4) mod 4 in
  let reponse_2 = if indcarte = 5 then (indjoueur+1) mod 4 else (indjoueur - indcarte/2 + 4) mod 4 in
  let reponse_3 =
    if indcarte < 3 then indjoueur
    else (indjoueur - indcarte + 6) mod 4 in
  [| reponse_0; reponse_1; reponse_2; reponse_3 |]
```

Question P2.6 :

```
let qui_appeler main =
  let cartes_en_main = Array.make 20 false in
  for i = 0 to Array.length main do
    match main.(i) with
    | Atout(n) when n < 21 -> cartes_en_main.(n-1) <- true
    | _ -> ()
  done;
  let reponse = ref 20 in
  while cartes_en_main.(!reponse-1) do decr reponse done;
  (* En fonctionnement normal, pas de risque d'indice qui déborde. *)
  !reponse
```

Question P2.7 :

```
let points_carte c = match c with
| Atout(1) | Atout(21) | Atout(22) -> 5
| Coul(Roi, _) -> 5
| Coul(Dame, _) -> 4
| Coul(Cavalier, _) -> 3
| Coul(Valet, _) -> 2
| _ -> 1;;

let points_plis tab = Array.fold_left (fun tot c -> tot + points_carte c) 0 tab
```

Question P2.8 :

On peut se forcer à vérifier que les quatre rois trouvés sont différents, par exemple avec un dictionnaire des couleurs, ou se contenter de compter sur la cohérence du tableau des cartes amassées.

```
let quatre_rois tab =
  let rois = Hashtbl.create 4 in
  for i = 0 to Array.length tab - 1 do
    match tab.(i) with
    | Coul(Roi, coul) -> Hashtbl.replace rois coul true
    | _ -> ()
  done;
  Hashtbl.length rois = 4
```

On a utilisé `replace` pour éviter qu'en cas d'incohérence dans le tableau on ne compte pas plusieurs fois un même roi, sachant que `replace` fonctionne comme `add` si la clé n'existe pas encore dans la table de hachage.

Question P2.9 :

```
let capture_21 plis camps =
  let localise_21 = ref (-1, -1) in
  let localise_excuse = ref (-1, -1) in
  for i = 0 to 8 do (* Array.length plis - 1 *)
    for j = 0 to 3 do (* Array.length plis.(i) - 1 *)
      match plis.(i).(j) with
      | Atout(21) -> localise_21 := (i, j)
      | Atout(22) -> localise_excuse := (i, j)
    done
  done;
  match !localise_21, !localise_excuse with
  | (i1, _), (i2, _) when i1 <> i2 -> 0
  | (_, j1), (_, j2) when camps.(j1) && not camps.(j2) -> -1
  | (_, j1), (_, j2) when not camps.(j1) && camps.(j2) -> 1
  | _ -> 0
```

Code golf

Solution personnelle :

```
s=lambda i:(i-1)%7%2==1
```

Solution d'Hugo D. (MPI*) :

```
s=lambda i:(-i)%7%2==0
```