Correction du DS 4

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 :

Imaginons que l'on ajoute un élément à un ABR presque complet de taille une puissance de deux moins deux, ce qui donne une taille égale à une puissance de deux moins un (l'arbre est alors complet), et que l'élément ajouté soit inférieur au minimum de l'ABR. Dans ce cas, pour respecter la structure d'ABR, chaque élément doit être modifié (on peut imaginer une insertion dans un tableau trié) et la complexité de l'opération sera nécessairement (au moins) linéaire en la taille de l'arbre.

Exercice 2 :

```
type 'a arbin = Vide | Noeud of 'a arbin * 'a * 'a arbin

let rec hauteur_arbre a = match a with
| Vide -> 0
| Noeud (g, _, d) -> 1 + max (hauteur_arbre g) (hauteur_arbre d)

let rec pow2 k = if k = 0 then 1 else 2 * pow2 (k-1)

let taille_tableau_arbre a = let h = hauteur_arbre a in pow2 h - 1

let arbin_vers_tableau a =
  let n = taille_tableau_arbre a in
  let reponse = Array.make n None in
  let rec remplit a ind = match a with
  | Vide -> ()
  | Noeud(g, etiq, d) -> reponse.(ind) <- Some etiq; remplit g (2*ind + 1); remplit d (2 * ind + 2)
  in remplit a 0; reponse

let tableau_vers_arbin t =
  let rec recupere ind = match t.(ind) with
  | None -> Vide
  | Some etiq -> Noeud(recupere (2*ind + 1), etiq, recupere (2*ind + 2))
  in recupere 0
```

Exercice 3 :

Le test de primalité sera assez bourrin et ne permettra pas de profiter des informations ailleurs, on aurait pu faire un crible vu les répétitions, mais cette correction est un speedrun.
Pour la fonction `semipremier`, le premier diviseur de `n` que l'on rencontre est forcément premier et il arrête la fonction, d'où l'absence de vérification sur `i`.

```
bool premier(int n)
{
  int i = 2;
  while (i * i <= n)
  {
    if (n % i == 0) return false;
  }
  return true;
}

bool semipremier(int n)
{
  if (n < 4) return false; // vu la fonction suivante
  int i = 2;
  while (i * i <= n)
  {
    if (n % i == 0)
    {
      return premier (n / i);
    }
  }
  return false;
}

int* les_semipremiers(int mini, int maxi)
{
  int* reponsetropgrande = malloc((maxi-mini+1)*sizeof(int));
  int taillereponse = 1;
  int spcsc = 0;
  for (int i = mini-1 ; i <= maxi+1 ; i += 1)
  {
    if (semipremier(i))
    {
      spcsc += 1;
      if (spcsc == 3)
      {
        reponsetropgrande[taillereponse] = i-1;
        taillereponse += 1;
      }
    }
    else spcsc = 0;
  }
  int* reponse = malloc(taillereponse * sizeof(int));
  for (int i = 0 ; i < taillereponse ; i += 1)
  {
    reponse[i+1] = reponsetropgrande[i];
  }
  reponse[0] = taillereponse;
  free(reponsetropgrande);
  return reponse;
}
```

Exercice 4 :

On compte simplement les blocs de `true` consécutifs et on additionne leur taille divisée par quatre (avec un arrondi par défaut qui est exactement ce que fait la division euclidienne).

```
let nb_min_objets tab =
  let rep = ref 0 in
  let truecsc = ref 0 in
  for i = 0 to Array.length tab - 1 do
    if tab.(i) then incr truecsc
    else (rep := !rep + !truecsc / 4; truecsc := 0)
  done; !rep + !truecsc / 4
```

Exercice 5 :

Comme on a intérêt à prendre tous les anti-objets, et qu'il n'y a pas d'objet dans les cases avec un anti-objet, il suffit d'adapter les booléens aux entiers et d'ajouter le nombre d'anti-objets.

```
let nb_min_objets_bis tab =
  let rep = ref 0 in
  let truecsc = ref 0 in
  for i = 0 to Array.length tab - 1 do
    if tab.(i) = -1 then decr rep;
    if tab.(i) = 1 then incr truecsc
    else (rep := !rep + !truecsc / 4; truecsc := 0) (* y compris si on a un -1 *)
  done; !rep + !truecsc / 4
```

Exercice 6 :

La programmation dynamique s'impose ici.

```
int nb_min_obj(int* tab, int taille)
{
  int* memoire = malloc((taille+1)*sizeof(int));
  memoire[0] = 0;
  for (int i = 1 ; i <= taille ; i += 1)
  {
    memoire[i] = memoire[i-1];
    for (int j = i-2 ; j >= 0 && j >= i-4 ; j -= 1)
    {
      if (memoire[i] > memoire[j])
      {
        memoire[i] = memoire[j];
      }
    }
    memoire[i] += tab[i-1];
  }
  int reponse = memoire[taille];
  free(memoire);
  return reponse;
}
```

Problème

Question P1 :

Si l'élément majoritaire existe, il est unique, car avec un deuxième élément majoritaire les nombres d'occurrences des deux additionnés donnent strictement plus que la taille de la séquence.

Question P2 :

```
int nb_occ(int* tab, int taille, int elt)
{
  int rep = 0;
  for (int i = 0 ; i < taille ; i += 1)
  {
    if (tab[i] == elt) rep += 1;
  }
  return rep;
}
```

Question P3 :

```
int* majoritaire(int* tab, int taille)
{
  for (int i = 0 ; i < taille ; i += 1)
  {
    int nbocc = nb_occ(tab, taille, tab[i]);
    if (nbocc > taille / 2) // ou 2 * nbocc > taille
    {
      int* rep = malloc(sizeof(int)); // On autoriserait return &tab[i] mais c'est hors programme !
      *rep = tab[i];
      return rep;
    }
  }
  return NULL;
}
```

La complexité est quadratique en la taille du tableau dans le pire des cas, car on appelle la fonction de comptage de complexité toujours linéaire pour chaque élément (jusqu'à trouver l'élément majoritaire ou terminer l'étude du tableau). Petit raffinement : on peut arrêter la boucle dans la fonction `majoritaire` à la moitié de la taille...

Question P4 :

```
let rec ecremer l = match l with
| [] | [_] -> l
| a::b::q -> if a = b then a::ecremer q else ecremer q

let rec ecremer_repeter l = match l with
| [] | [_] -> l
| _ -> ecremer_repeter (ecremer l)

let verifie_majoritaire e l =
  let rec comptage n l = match l with
  | [] -> n > 0
  | a::q when a = e -> comptage (n+1) q
  | _::q -> comptage (n-1) q
  in comptage 0 l
```

```
let majoritaire l = match ecremer_repeter l with
| [a] when verifie_majoritaire a l -> Some a
| _ -> None
```

Question P5 :

En premier, il faut prouver la terminaison. Pour la fonction `ecremer`, on a pour pseudo-variant la taille de la liste, et on peut prouver par induction sur cette taille que la taille du résultat est au plus la moitié de la taille de la liste arrondie par excès, ce qui prouver que la taille de la liste est un pseudo-variant pour `ecremer_repeter` car au plus la moitié arrondie par excès, c'est strictement moins qu'une valeur supérieure ou égale à deux (les cas déclenchant une récursion). La fonction `verifie_majoritaire` termine car la fonction `comptage` a aussi la taille de la liste comme pseudo-variant, et la fonction `majoritaire` déclenche uniquement deux fonctions qui terminent.

À partir de cela, on trouve les formules de récurrence suivantes pour les complexités :
- `ecremer` : $c_n = c_{n-2} + \Theta(1)$, d'où $c_n = \Theta(n)$
- `ecremer_repeter` : $c_n = c_k + \Theta(n)$ avec un certain $k \leq \lceil \frac{n}{2} \rceil$
- `comptage` : $c_n = c_{n-1} + \Theta(1)$, d'où $c_n = \Theta(n)$ aussi.

Dans `ecremer_repeter`, la valeur de k est la taille de la liste obtenue par `ecremer`. Dans le pire des cas, c'est la moitié de la taille de la liste avec un arrondi supérieur. Le théorème de récurrence des partitions (ou la liste des cas classiques avec une légère adaptation due à l'arrondi par excès) nous donne une complexité linéaire dans le pire des cas.

Conclusion : en ajoutant tout ceci, `majoritaire` est en $\Theta(n)$.

Question P6 :

On choisit le tri fusion pour le plaisir de le faire lui aussi sur des tableaux.

Cette version ne mute pas l'argument et est de complexité $\Theta(n \log n)$ où n est la taille du tableau.

```
let separe tab =
  let n = Array.length tab in
  (Array.sub tab 0 (n/2), Array.sub tab (n/2) (n-n/2))

let fusionne tab1 tab2 =
  let n1 = Array.length tab1 in
  let n2 = Array.length tab2 in (* potentiellement différents d'une unité *)
  let n = n1 + n2 in
  let rep = Array.make n (tab1.(0)) in (* la fonction n'est pas appelée si un tableau peut être vide *)
  let i1 = ref 0 in
  let i2 = ref 0 in
  for i = 0 to n - 1 do
    if !i1 = n1 then (rep.(i) <- tab2.(!i2); incr i2)
    else if !i2 = n2 || tab1.(!i1) < tab2.(!i2) then (rep.(i) <- tab2.(!i1); incr i1)
    else (rep.(i) <- tab2.(!i2); incr i2)
  done;
  rep

let rec tri_fusion tab =
  if Array.length tab < 2 then Array.copy tab
  else let t1, t2 = separe tab in
  fusionne (tri_fusion t1) (tri_fusion t2)
```

Question P7 :

```
let premier_indice tab e = (* si on appelle la fonction, e est dans tab *)
  let i = ref 0 in
  while tab.(!i) <> e do incr i done;
  !i

let majoritaire tab =
  let n = Array.length tab in
  let tabtrie = tri_fusion tab in
  let i = ref 0 in
  let nbid = ref 1 in
  while !nbid <= n / 2 && !i < n-1 do
    if tabtrie.(!i) = tabtrie.(!i+1) then incr nbid
    else nbid := 1;
    incr i
  done; if !nbid > n / 2 then premier_indice tab (tabtrie.(!i)) else -1
```

Question P8 :

```
let majoritaire l =
  let dico = Hashtbl.create 8 in
  let rec compte_et_compte l = match l with
  | [] -> 0
  | a::q ->
    begin
      match Hashtbl.find_opt dico a with
      | None -> Hashtbl.add dico a 1
      | Some n -> Hashtbl.replace dico a (n+1)
    end;
    1 + compte_et_compte q
  in let taille = compte_et_compte l in
  Hashtbl.fold (fun a n acc -> if n > taille / 2 then Some a else acc) dico None
```

Question P9 :

Si un élément est majoritaire, il est en particulier médian parmi les valeurs dans l'ordre. Comme on peut calculer la médiane en temps linéaire par l'algorithme « médiane des médianes », on peut déterminer en temps linéaire l'unique candidat à être l'élément majoritaire, puis vérifier par un comptage comme dans le cas d'autres algorithmes de ce problème que le candidat est majoritaire, le tout sera alors en temps linéaire (et en espace linéaire au vu des ressources consommées par l'algorihme de calcul de la médiane).

Question P10 :

```
int* boyer_moore(int* tab, int taille)
{
  if (taille < 0) return NULL;
  int candidat = tab[0];
  int comptage = 1;
  for (int i = 1 ; i < taille ; i += 1)
  {
    if (comptage == 0)
    {
      candidat = tab[i];
      comptage = 1;
    }
    else if (tab[i] == candidat) comptage += 1;
    else comptage -= 1;
  }
  if (comptage > 0)
  {
    int nbocc = nb_occ(tab, taille, candidat); // Recyclage de P2
    if (nbocc > taille / 2) // ou 2 * nbocc > taille
    {
      int* rep = malloc(sizeof(int)); // On autoriserait return &tab[i] mais c'est hors programme !
      *rep = tab[i];
      return rep;
    }
  }
}
```

Question P11 :

On se sert de l'invariant de boucle suivant pour le traitement effectué pendant le parcours (qu'on peut suivre au niveau de la boucle de la fonction écrite ci-avant) : « Après avoir étudié `i` éléments, il y a au moins `comptage` occurrences du candidat et tous les autres éléments peuvent être regroupés par paires de nombres distincts (entre deux éléments d'une même paire) ».

Les trois cas de figure préservent l'invariant (avec les nouvelles valeurs des variables après la mise à jour effectuée par le tour de boucle) qui est par ailleurs initialement vrai.

Le fait qu'il soit vrai en sortie de boucle garantit qu'aucun élément à part le candidat ne peut être majoritaire car il serait toujours apparié à un élément différent en-dehors des `comptage` valeurs qui sont en plus différentes de l'élément qui ne serait pas le candidat.