TD de *Logique et Circuits* n° 10
(Correction)

Induction sur les listes

Exercice 1 Le schéma de définition inductive sur les listes vu en cours est:

$$f(l) = g(l, f(h(l)))$$

ou, pour des fonctions à $k+1$ arguments:

$$f(l, a_1, ..., a_k) = g(l, a_1, ..., a_k, f(h(l), i_1(a_1)..., i_k(a_k)))$$

.

Pour chaque fonction Ocaml de la liste suivante, donner l'équation correspondante:

1. ```
   let rec parite liste=
     match liste with
       [] -> true
     |a::l-> not(parite l);;
   ```

2. ```
   let rec membre a l =
     match l  with
       []    -> false
     | b::j  -> a=b || membre a j;;
   ```

3. ```
   let rec somme_impair liste=
     match liste with
       [] -> 0
     | [a]-> a
     | a::b::k-> a + somme_impair k
   ```

Correction :

1. $g(l, b) = \neg b$
   $h(a :: l) = l$
   $f([]) = true$
2. $g(a, b :: l, c) = (a = b) \vee c$
   $i_1(a) = a$
   $h(a :: l) = l$
   $f(a, []) = false$
3. $g(a :: b :: l, s) = a + s$
   $h(a :: b :: l) = l$
   $f([]) = 0$
   $f(a :: []) = a$
   ```

Exercice 2 Le principe du "tri par fusion" (*merge-sort*) est le suivant: pour trier une liste l, on commence par construire les deux sous-listes l_p et l_i, qui contiennent respectivement les éléments d'indice pair et impair de l (cette opération est appelée ici "séparer").

On trie (par fusion) l_p et l_i, puis on fusionne les deux listes triées l_p^{triee} et l_i^{triee}, obtenant ainsi l^{triee}.

On suppose disposer d'une implantation en Ocaml des fonctions de séparation et fusion:

```
separer:  'a liste -> ('a liste * 'a liste)
fusion:  ('a liste * 'a liste)-> 'a liste
```

1. Ecrire en Ocaml la fonction `tri_fusion: 'a liste -> 'a liste`.
2. Vérifier que `tri_fusion` n'appartient pas à l'ensemble de fonctions définissables avec le schéma $f(l) = g(l, f(h(l))$, et proposer un schéma plus général, qui inclut des fonctions comme `tri_fusion` (ou `tri_rapide`).
3. Calculer la complexité de `tri_fusion` en termes du nombre de comparaisons, sachant que le nombre de comparaisons nécessaires pour fusionner deux listes de longueur n est au mieux n, au pire $2n-1$, et que l'opération de séparation ne fait pas de comparaisons.

Correction :

1.
```
let rec tri_fusion l = match l with
  [] -> []
| [a] -> l
| _  -> let l1,l2 = separer l in fusion (tri_fusion l1) (tri_fusion l2)
;;
```
2. En effet, le double appel récursif echappe au schéma simple. On peut faire:
   $f(l) = g(l, f(h_1(l)), f(h_2(l)))$
   Ceci est suffisant pour `tri_fusion` et `tri_rapide`. Plus généralement,
   $f(l) = g(l, f(h_1(l)), ..., f(h_n(l)))$
   ou, avec un schéma unique:
   $f(l) = g(l, map\ f\ liste\_des\_h_i(l))$
3. Soit $n = longueur(l) = 2^k$. Pendant le calcul de `tri_fusion` l, on fusionne 2^{k-1} paires de listes de longueur 1, 2^{k-2} paires de listes de longueur 2, ..., 1 paire de listes de longueur 2^{k-1}.
   Donc, le nombre total de comparaisons est au mieux:
   $\sum_{i=0}^{k-1} 2^{k-1-i}2^i = k2^{k-1}$
   au pire:
   $\sum_{i=0}^{k-1} 2^{k-1-i}(2^{i+1}-1) = k2^k - \sum_{i=0}^{k-1} 2^{k-1-i} = k2^k - 2^k + 1 = (k-1)2^k + 1$
   dans les deux cas, la complexité est $O(n \times log_2(n))$

Exercice 3 L'intersection de deux listes sans répétitions l_1 et l_2 est une liste qui contient les éléments qui apparaissent à la fois dans l_1 et dans l_2.

Une fonction auxiliare utile pour programmer l'intersection est `membre: 'a -> 'a list -> bool` (exercice 1).

1. Ecrire la fonction Ocaml `intersection: 'a list -> 'a list -> 'a list`, qui réalise l'équation:

   $f(l_1, l_2) = g(l_1, l_2, f(h(l_1), i_1(l_2)))$

   $g(a :: l_1, l_2, r) = \begin{cases} a :: r & \text{si } \texttt{membre}(a, l_2) \\ r & \text{sinon} \end{cases}$

   $h(a :: l) = l$

   $i_1(l) = l$

Pour calculer l'intersection de deux listes triées, il n'est pas nécessaire d'utiliser le test d'appartenance.

2. Ecrire la fonction

   `intersection2: ('a list * 'a list) -> 'a list`

   utilisant `tri_fusion` et n'utilisant pas `membre`.

3. Comparer les complexités de `intersection` et `intersection2`, en termes du nombre de comparaisons et tests d'égalité (qu'on ne distingue pas).

Correction :

1. ```
   let rec intersection l j = match l with
   [] -> []
   |a::k -> let res = intersection k j  in
       if ((membre a j)  then a::res
                         else res;;
   ```

2. ```
   let rec intersection2 l j = let lt=tri_fusion l and jt=tri_fusion j in
   match lt,jt with
   [],k->[]
   |k,[]->[]
   |a::k,b::i -> if a=b then a::intersection2 k i else
                   if a<b then intersection2 k (b::i) else
                       intersection2 (a::k) i;;
   ```

3. Soient $n = longueur(l_1)$, $m = longueur(l_2)$. `intersection` l_1 l_2 comporte n appels de `membre` a l_2. Chaque appel coute au miex 1, au pire m. Donc la complexité de `intersection` est au mieux $O(n)$, au pire $O(n \times m)$.

   Une fois trié les listes, `intersection2` execute au mieux $min(m, n)$ tests, au pire $2min(m, n)$ tests, donc la complexité de `intersection2` est $O(n \times log_2(n) + m \times log_2(m)) = O(max(n, m) \times log_2(max(n, m)))$

Exercice 4 Prouver la proposition suivante: Pour toute liste l `intersection` l $l = l$.

Correction : Il faut prouver plus généralement que, si l est une sous-liste de l alors `intersection` l l' $= l$

(formellement, l sous-liste de l' s'écrit

$\exists f : \{0, .., longueur(l) - 1\} \to \{0, .., longueur(l') - 1\}$ telle que $\forall\ 0 \leq i \leq longuer(l) - 1$ $elt(l, i) = elt(l', f(i))$

où $elt(a :: j, 0) = a$, $elt(a :: j, n + 1) = elt(j, n)$

Cet énoncé se demontre facilement par induction, avec la remarque que si $l = a :: j$ et f envoi l sur l' au sens de la définition ci-dessus, alors $n \mapsto f(n + 1)$ envoi j sur l'.