# Devoir surveillé n° 1 — corrigé

## III Représentation d'ensembles avec des listes

III.1) Par la liste vide bien sûr.

III.2) Il suffit de tester si l'ensemble est la liste vide. Cette fonction s'exécute en temps constant.

OCaml
```
let est_vide ens = ens = []
```

III.3) C'est la fonction `List.length` de complexité linéaire en la taille $n$ de la liste car vérifie $T(n) = T(n-1) + \Theta(1)$ et $T(0) = \Theta(1)$.

OCaml
```
let rec cardinal ens =
  match ens with
  | [] -> 0
  | _ :: suite -> 1 + cardinal suite
```

III.4) C'est la fonction `List.mem` que l'on écrit élégamment en utilisant le caractère paresseux de l'opérateur booléen. Sa complexité dans le pire cas est linéaire en la taille de la liste car elle vérifie aussi $T(n) = T(n-1) + \Theta(1)$ et $T(0) = \Theta(1)$.

OCaml
```
let rec appartient elt ens =
  match ens with
  | [] -> false
  | obj :: suite -> (obj = elt) || appartient elt suite
```

III.5) Il faut bien veiller à ne pas ajouter de doublons. La complexité est linéaire en la taille de la liste : c'est celle de la fonction précédente plus un temps constant.

OCaml
```
let ajoute elt ens =
  if appartient elt ens then
    ens
  else
    elt :: ens
```

III.6) On pourrait encore utiliser la fonction `appartient`, on aurait alors deux parcours de la liste si l'élément est présent, mais on éviterait de reconstruire inutilement la liste si ce n'est pas le cas. Dans tous les cas la complexité dans le pire cas est linéaire en la taille de la liste. Dans le deuxième cas on sait que l'élément n'est plus présent dans la suite puisque la liste est sans doublons.

OCaml
```
let rec supprime elt ens =
  match ens with
  | [] -> []
  | obj :: suite when obj = elt -> suite
  | obj :: suite -> obj :: (supprime elt suite)
```

III.7) La fonction `intersection : ensemble -> ensemble -> ensemble`, ajoute successivement tous les éléments du premier ensemble qui appartiennent aussi au deuxième. Il y a exactement $|ens_1|$ appels récursifs, chacun de complexité $\Theta(|ens_2|)$. La complexité est donc en $\Theta(|ens_1||ens_2|)$.

OCaml
```
let rec intersection ens1 ens2 =
  match ens1 with
  | [] -> []
  | elt1 :: suite1 when appartient elt1 ens2 ->
      elt1 :: (intersection suite1 ens2)
  | _ :: suite1 ->
      intersection suite1 ens2
```

III.8) La fonction `union : ensemble -> ensemble -> ensemble` ajoute au deuxième ensemble tous les éléments du premier qui ne sont pas dans le deuxième. Sa complexité est aussi en $\Theta(|ens_1||ens_2|)$.

OCaml
```
let rec union ens1 ens2 =
  match ens1 with
  | [] -> ens2
  | elt1 :: suite1 when appartient elt1 ens2 ->
      union suite1 ens2
  | elt1 :: suite1 ->
      elt1 :: (union suite1 ens2)
```

III.9) On peut implémenter une fonction `inclusion : ensemble -> ensemble -> bool` et vérifier la double inclusion. On utilise ici directement les fonctions précédentes pour implémenter la fonction `egal : ensemble -> ensemble -> bool`. Dans tous les cas, la complexité est en $\Theta(|ens_1||ens_2|)$.

OCaml
```
let egal ens1 ens2 =
  let c = cardinal (intersection ens1 ens2) in
  c = (cardinal ens1) && c = (cardinal ens2)
```