Devoir surveillé n° 3
Éléments de correction

Problème 1 - D'après CCP 2010

1.
```
let disjoints i1 i2 =
  let a1, b1 = i1 in
  let a2, b2 = i2 in
  b1 < a2 || b2 < a1
;;
```

2.
```
let fusion i1 i2 =
  let a1, b1 = i1 in
  let a2, b2 = i2 in
  (min a1 a2), (max b1 b2)
;;
```

3. Soit $[i_1, \dots ; i_n]$ une liste triée d'intervalles bien formés. Soient j et k deux entiers de $[\![1, n]\!]$ tels que $j < k$. Soient m_j, M_j, m_k, M_k tels que $i_j = [m_j, M_j]$ et $i_k = [m_k, M_k]$. i_j et i_k sont bien formés et la liste est triée, donc $i_j < i_k$. Par conséquent, $m_j :\leqslant M_j < m_k \leqslant M_k$, donc i_j et i_k sont disjoints.
   Finalement, les intervalles d'une liste triée d'intervalles bien formés sont deux à deux disjoints.

4. Pour un arbre a, notons $\mathcal{P}(a)$ la propriété : pour tous les nœuds de l'arbre a, le maximum de l'intervalle englobant du fils gauche est strictement plus petit que le minimum de l'intervalle englobant du fils droit, alors l'arbre est bien formé.
   On démontre par récurrence que pour tout $n \in \mathbb{N}$, pour tout arbre de hauteur inférieure ou égale à n, si $\mathcal{P}(a)$, alors a est bien formé.

   Initialisation. Si a est un arbre d'intervalles bien formés de hauteur 0, a est une feuille contenant un intervalle bien formé $[min_a, max_a]$, donc l'arbre a est bien formé.

   Hérédité. Soit $n \in \mathbb{N}$. On suppose la propriété vérifiée par les arbres d'intervalles bien formés de hateur au plus n. Soit a un arbre d'intervalles bien formés de hauteur $n+1$ et vérifiant $\mathcal{P}(a)$. Alors son sous-arbre gauche $\mathcal{G}(a)$ et son sous-arbre droit $\mathcal{D}(a)$ ont des hauteurs inférieures ou égales à n et vérifient la propriété $\mathcal{P}$.
   D'après l'hypothèse de récurrence, $\mathcal{G}(a)$ et $\mathcal{D}(a)$ sont des arbres bien formés, donc $\langle f_1, \dots, f_m \rangle = \mathcal{E}(\mathcal{G}(a))$ et $\langle f_{m+1}, \dots, f_{m+p} \rangle = \mathcal{E}(\mathcal{D}(a))$ sont des listes triées.
   Comme l'intervalle englobant $\mathcal{G}(a)$ est strictement plus petit que l'intervalle englobant $\mathcal{D}(a)$, f_m est strictement plus petit que f_{m+1} et donc $\langle f_1, \dots, f_m, f_{m+1}, \dots, f_{m+p} \rangle$ est une liste triée d'intervalles bien formés (qui sont donc deux à deux disjoints).
   Finalement, a est un arbre bien formé.

   Conclusion. Tout arbre a d'intervalles bien formés vérifiant $\mathcal{P}(a)$ est un arbre bien formé.

5. En s'appuyant sur le parcours gauche-droite : si $\langle f_1, \dots, f_n \rangle$ est le résultat du parcours gauche-droite de l'arbre,

$$ABFI(a) \Leftrightarrow \forall k \in [\![1, n]\!],\ \min f_k \leqslant \max f_k \text{ et } \forall k \in [\![1, n]\!],\ \max f_k < \min f_{k+1}$$

   En s'appuyant sur la structure d'arbre

$$ABFI(a) \Leftrightarrow (a \in \mathcal{F} \text{ et } \min a \leqslant \max a)$$
$$\text{ou } (a \in \mathcal{N} \text{ et } ABFI(\mathcal{G}(a)) \text{ et } ABFI(\mathcal{D}(a)) \text{ et } \max(\mathcal{G}(a)) < \min(\mathcal{D}(a)))$$

6.
```
let rec englobant a =
  match a with
  | Vide -> failwith "Arbre vide"
  | Feuille (d, f) -> d,f
  | Noeud (fg, fd) -> fusion (englobant fg) (englobant fd)
;;
```

7. Le fait de ne devoir parcourir qu'une fois l'arbre oblige à écrire une fonction auxiliaire récursive prenant en argument un arbre et renvoyant :
   - un booléen indiquant s'il s'agit d'un arbre bien formé ;
   - le plus petit entier contenu dans les intervalles de l'arbre s'il est bien formé ;
   - le plus grand entier contenu dans les intervalles de l'arbre s'il est bien formé.

```
let verifier a =
  let rec aux a =
    match a with
    | Vide -> true, max_int, min_int
    | Feuille (mi, ma) -> mi <=  ma, mi, ma
    | Noeud (fg, fd) ->
       let bf_g, mi_g, ma_g = aux fg in
       let bf_d, mi_d, ma_d = aux fd in
       bf_g && bf_d && ma_g < mi_d, mi_g, ma_d
  in
  let bf, _, _ = aux a in
  bf
;;
```

8.
```
let rec appartenir v a =
  match a with
  | Vide -> false
  | Feuille (mi, ma) -> mi <= v && v <= ma
  | Noeud (fg, fd) -> (appartenir v fg) || (appartenir v fd)
;;
```

9. On commence par écrire une fonction `fusion_abf` tel que l'appel sur deux arbres bien formés a_1 et a_2 tels que $\max a_1 < \min a_2$ renvoie un arbre bien formé contenant les intervalles de a_1 et a_2.

```
let fusion_abf a1 a2 =
  match a1, a2 with
  | Vide, _ -> a2
  | _, Vide -> a1
  | _, _ -> Noeud(a1, a2)
;;

let rec decouper v a =
  match a with
  | Vide -> Vide, Vide, Vide
  | Feuille (mi, ma) ->
     if v < mi
     then Vide, Vide, a
     else
       if v > ma
       then a, Vide, Vide
       else Vide, a, Vide
  | Noeud (fg, fd) ->
     let ag, bg, cg = decouper v fg in
     let ad, bd, cd = decouper v fd in
     fusion_abf ag ad, fusion_abf bg bd, fusion_abf cg cd
;;
```

10. On écrit une fonction auxiliaire récursive, qui à un intervalle i et un arbre bien formé a associe un triplet constitué :
    - d'un arbre bien formé contenant les intervalles de a strictement plus petits que i ;
    - de la fusion de i avec les intervalles de a qui ne sont pas disjoints de i ;
    - d'un arbre bien formé contenant les intervalles de a strictement plus grand que i.

```
let ajouter i a =
  let rec aux i  a =
    match a with
    | Vide -> Vide, i, Vide
    | Feuille (mi, ma) ->
       let c, d = i in
       if ma < c
       then a, i, Vide
       else
         if mi > d
         then Vide, i, a
         else Vide, fusion i (mi, ma), Vide
    | Noeud (fg, fd) ->
       let ag, ig, bg = aux i fg in
       let ad, id, bd = aux i fd in
       fusion_abf ag ad, fusion ig id, fusion_abf bg bd
  in
  let fg, (c, d), fd = aux i a in
  fusion_abf (fusion_abf fg (Feuille (c,d))) fd
;;
```

11. Les fonctions `fusion` et `fusion_abf` s'exécutent en temps $O(1)$. La fonction `ajouter` visite une et une seule fois chacun des nœuds de l'arbre donc sa complexité est en $O(n)$ où n est le nombre de nœuds de l'arbre (et ne dépend pas de `i`).

    Le meilleur des cas correspond donc au cas d'un arbre « peigne » ayant une feuille à chaque niveau, qui comporte $2h+1$ nœuds (en considérant que la profondeur de la racine est nulle). La complexité dans le meilleur des cas de la fonction `ajouter` est donc en $O(h)$.

    Le pire des cas correspond au cas d'un arbre binaire complet, qui comporte $2^{h+1}-1$ nœuds, donc la et la complexité dans le pire des cas est en $O(2^h)$.

Problème 2 - D'après Centrale 2018

1.
```
let dichotomie a t =
  let i = ref 0 and j = ref (Array.length t) in
  while !i < !j - 1 do
    let m = (!i + !j) / 2 in
    if t.(m) > a
    then j := m
    else i := m
```

```
    done;
    !i
  ;;
```

2.
```
let deplacements_grille (a,b) =
  let ob_lig = obstacles_lignes.(a) in
  let ob_col = obstacles_colonnes.(b) in
  let i = dichotomie b ob_lig and j = dichotomie a ob_col in
  [|(a, ob_lig.(i)); (a, ob_lig.(i+1)-1); (ob_col.(j),b); (ob_col.(j
+1)-1,b); |]
;;
```

3.
```
let matrice_deplacements () =
  let m = Array.make_matrix n n [||] in
  for a = 0 to n - 1 do
    for b = 0 to n - 1 do
      m.(a).(b) <- deplacements_grille (a, b)
    done
  done;
  m
;;
```

4.
```
let modif t (a,b) (c,d) =
  if a = c
  then
    begin
      if d < b
      then
        let _, w = t.(0) in
        t.(0) <- a, max w (d+1);
      else
        let _, e = t.(1) in
        t.(1) <- a, min e (d-1);
    end;
  if b = d
  then
    begin
      if c < a
      then
        let n, _ = t.(2) in
        t.(2) <- max n (c+1), b
        ;
      else
        let s, _ = t.(3) in
        t.(3) <- min s (c-1), b
    end
;;
```

5.
```
let deplacements_robots (a, b) q =
  let t = Array.copy mat_deplacements.(a).(b) in
  let rec aux lst =
    match lst with
    | [] -> ()
    | (c, d) :: q -> modif t (a, b) (c, d);
                     aux q
  in
  aux q;
  t
;;
```

6. Pour chaque déplacement, on peut déplacer un des quatre robots dans les quatre directions ; il y a donc 16 déplacements possibles. Par conséquent, le nombre de suites de k déplacements est 16^k. Le résultat de chaque déplacement se calcule en utilisant la fonction `deplacements_robots` qui s'exécute en temps constant (le nombre de robots étant fixé).

   Par ailleurs, la construction de `mat_deplacements` nécessite un appel à la fonction `matrice_deplacements`, qui réalise n^2 appels à `deplacements_grille`. Chaque appel à `deplacements_grille` effectue deux appels à `dichotomie`, en temps $O(\log n)$, et des opérations en temps constant. Par conséquent, la omplexité de la construction de `mat_deplacements` est $O(n^2 \log n)$.

   La complexité de cette approche est donc un $O(n^2 \log n + 16^k)$.

7.
```
let rec insertion x q =
  match q with
  | [] -> [x]
  | hd::tl -> if x < hd
      then x::q
      else hd::(insertion x tl)
;;
```

8.
```
let rec tri_insertion q =
  match q with
  | [] -> []
  | hd::tl -> insertion hd (tri_insertion tl)
;;
```

9. Dans le meilleur des cas, l'insertion se fait toujours en tête (cas d'une liste déjà triée). La complexité dans le meilleur des cas de ce tri est linéaire.
   Dans le pire des cas, l'insertion se fait en fin de liste (cas d'une liste triée dans l'ordre décroissant). La complexité dans le pire des cas de ce tri est quadratique.

   Soit $L = [a_0; \dots; a_k; \dots; a_{n-1}]$ une liste telle que seul a_k n'est pas à sa place. Alors :
   - Lors des appels récursifs, l'insertion des éléments $a_{k+1}, \dots, a_{n-1}$ s'effectue en $O(1)$, car ces éléments sont déjà dans l'ordre.
   - Si $a_k > a_{k+1}$, alors l'insertion de a_k s'effectue en $O(n)$, puis l'insertion des éléments $a_0; \dots; a_{k-1}$ s'effectue en temps $O(1)$ (car ces éléments sont plus petits que a_{k+1}.
   - Si $a_k < a_{k-1}$, alors l'insertion de a_k s'effectue en $O(1)$ (car en tête de liste), puis chacun des éléments $a_0; \dots; a_{k-1}$ s'insère soit avant a_k, soit juste après a_k, donc en temps $O(1)$.

   Dans tous les cas, le tri de la liste est alors linéaire.

10.
```
let rec mem1 x q =
  match q with
  | [] -> false
  | (a, _)::tl -> a = x || mem1 x tl
;;
```

11.
```
let rec assoc x q =
  match q with
  | [] -> failwith "Not Found"
  | (a, b)::tl ->
     if a = x
     then b
     else assoc x tl
;;
```

12. On utilise l'algorithme de Hörner.

```
let rec hachage_liste w q =
  match q with
  | [] -> 0
  | (a, b)::tl -> (((hachage_liste w tl)*n + b)*n + a) mod w
;;
```

13.
```
let creer_table h w =
{ hache = h; donnees = Array.make w []; largeur = w }
;;
```

14.
```
let recherche t k =
  let alveole = t.donnees.(t.hache k) in
  mem1 k alveole
;;
```

15.
```
let element t k =
  let alveole = t.donnees.(t.hache k) in
  assoc k alveole
;;
```

16.
```
let ajout t k e =
  let alveole = t.donnees.(t.hache k) in
  if not mem1 k alveole
  then t.donnees.(t.hache k) <- (k, e)::alveole
;;
```

17.
```
let suppression t k =
  let hk = t.hache k in
  let alveole = t.donnees.(hk) in
  let rec suppr lst k =
    match lst with
    | [] -> []
    | (a, b)::tl -> if a = k
                    then tl
                    else (a, b)::(suppr tl k)
  in
  t.donnees.(hk) <- suppr alveole k
;;
```

18. La recherche d'une clé dans une alvéole est linéaire en la taille de l'alvéole. Sous l'hypothèse de hachage uniforme, l'espérance de la complexité de la recherche de k est la taille moyenne des alvéoles, qui est égale à $\alpha = n/w$. Par conséquent, l'espérance de la complexité de la recherche de k est bien un $O(1 + \alpha)$.

19. Soit k une clé présente dans la table. Chacune des autres clés ayant une probabilité de se trouver dans la même alvéole que k, l'espérance de la taille de l'alvéole de k est $1 + \frac{n-1}{w}$, donc la recherche de la clé k se fait bien en $O(1+\alpha)$.