Localisation de Point - Correction

Samy Jaziri

1 Introduction

Génération aléatoire d'ensembles

Pas de difficulté ici

```
(* Paramètre : n entier positif *)
(* Génère u_n tel que décrit dans l'énoncé *)
let rec u ( n : int ) : int =
  if n == 0 then u0 else 37698 * ( u ( n - 1) ) mod 524287 ;;
```

On construit les ensembles E_n comme décrit dans l'énoncé.

```
(* Paramètre = n : entier positif *)
(* Génère E_n tel que décrit dans l'énoncé *)
let rec e ( n : int ) : point ensemble =
  let p : point = (0, 0) in
  let ensemble : point ensemble = Array.make n p in
  for i = 0 to n - 1 do
    ensemble.(i) <- ( 2 * ( u ( 2 * i ) ) , 2 * ( u ( 2 * i + 1 ) ) )
  done ;
  ensemble ;;
```

2 Recherche logarithmique

Recherche Dichotomique dans un Tableau

On implémente la recherche dichotomique demandée par l'énoncé. On demande ici de renvoyer le plus grand élément de l'ensemble, inférieur ou égal à l'élément recherché.

- Si l'élément est égal à un élément de l'ensemble on renvoie celui-ci.
- Si l'élément recherché n'est pas égal à un élément de l'ensemble, on renvoie le dernier élément considéré par une recherche dichotomique qui conserve le pivot si ce dernier est plus petit que l'élément recherché.
- Si l'élément recherché est plus petit que tous les éléments de l'ensemble, l'algorithme doit renvoyer l'élément passé en paramètre de la fonction, faisant office de borne inférieur de l'espace de recherche.

Abstraitement, l'algorithme de recherche travaillera sur le tableau représentant l'ensemble auquel est ajouté la borne inférieure en première position (indice -1).

```
open Kholle_localisation_de_point ;;

let recherche_dichotomique
    ( comp : 'a -> 'b -> ordre )
    ( a : 'a ) ( inf : 'b )
    ( e : 'b ensemble ) : 'b =
  let rec recherche_bornee ( debut : int ) ( fin : int ) : 'b =
    if debut = fin then ( if debut = -1 then inf else e.(debut) )
    else let milieu = ( debut + fin + 1 ) / 2 in
      match comp a e.(milieu) with
      | PLUS_PETIT -> recherche_bornee debut ( milieu - 1 ) (* I *)
      | PLUS_GRAND -> recherche_bornee milieu fin (* II *)
      | EGAUX -> e.(milieu) in (* III *)
  recherche_bornee ( -1 ) ( ( Array.length e ) - 1 ) ;;
```

Avec une implémentation récursive de la boucle de recherche la preuve de correction peut sembler un peu plus complexe (bien qu'il s'agisse que d'une gymnastique mentale). Il est préférable de faire une implémentation itérative et de prouver la correction à l'aide d'un invariant de boucle si vous n'êtes pas à l'aise avec ces preuves.
On montre la correction de la fonction `recherche_bornee` par récurrence forte sur la taille du sous-tableau étudié (`fin - debut`). La propriété que l'on montre est la suivante : /En considérant que `inf` est à l'indice `-1` du tableau, si le plus grand élément de `e` plus petit que `a` est situé entre les indice `debut` et `fin`, alors la fonction `recherche_bornee` renvoie se dernier=.
Cas `debut = fin` : L'algorithme renvoie le seul élément du sous-tableau qui par hypothèse est le bon élément.
Cas `fin - debut = n+1` : Supposons que la propriété est vrai pour tout sous-tableau de taille inférieure ou égale à n. On distingue les trois cas du pattern matching :

- **I** : Comme le tableau est trié, la propriété **(b)** de `comp` permet d'affirmer que tous les éléments du tableau situés à gauche du milieu sont plus grand que **(a)**. Le plus grand élément de `e` plus petit que `a` est donc situé entre l'indice `debut` et l'indice `milieu - 1`. Par hypothèse de récurrence, l'appel de la fonction entre `debut` et `milieu - 1` renvoie l'élément recherché.
- **II** : Idem mais en utilisant **(b)**
- **III** : Les propriétés **(c)** et **(d)** permettent d'éliminer la possibilité pour un élément plus grand que le milieu d'être aussi égal à `a`, on peut donc renvoyer l'élément du milieu.

On obtient la correction par récurrence forte.
Pour chercher les éléments dans E_n demandés à la question 4 on défini la fonction suivante :

```
let recherche_e_n ( e : point ensemble ) ( x : int ) : point =
  let comp ( x : int ) ( p : point ) : ordre =
    let x', _ = p in ordre_int x x' in
  recherche_dichotomique comp x ( 0, 0 ) e ;;
```

Recherche dans un Arbre Binaire de Recherche

On implémente ensuite une recherche analogue dans un arbre binaire de recherche. On descend dans l'ABR en branchant à chaque nœud, à gauche si l'élément recherché est plus petit que ledit nœud, à droite si l'élément est plus grand. Dans le cas d'égalité, on s'arrête. À chaque descente à droite, on se souvient du nœud de ce branchement, c'est la plus grande des valeurs inférieures à l'élément recherché que l'on a rencontré jusque-là.

```
let recherche_abr
    ( comp : 'a -> 'b -> ordre )
    ( a : 'a ) ( inf : 'b )
    ( abr : 'b abr ) : 'b =
  let rec parcours ( sous_abr : 'b abr ) ( element_inf : 'b ) : 'b =
    match sous_abr with
    | Vide -> element_inf
    | Noeud ( e, abr_g, abr_d ) ->
      match comp a e with
      | PLUS_PETIT -> parcours abr_g element_inf
      | PLUS_GRAND -> parcours abr_d e
      | EGAUX -> e in
  parcours abr inf ;;
```

Correction : On rappelle qu'un prérequis à l'algorithme est que `inf` soit plus petit que `a` et tous les éléments de `abr`. Pour tout ABR $\mathcal{A}$ on note $\mathcal{E}(\mathcal{A})$ l'ensemble des étiquettes des nœuds de $\mathcal{A}$.
Encore une fois la preuve de correction est plus difficile à écrire avec une implémentation recursive. On prouve par induction sur `sous-abr` que **si**

- `element_inf` est le plus grand des éléments inférieurs à `a` parmi $\{\texttt{inf}\}\cup(\mathcal{E}(\texttt{abr})\backslash\mathcal{E}(\texttt{sous\_abr}))$ **(1)**
- **et** $\forall e \in \mathcal{E}(\texttt{sous\_abr}), \texttt{e} > \texttt{element\_inf}$ **(2)**

alors `parcours sous_abr element_inf` renvoie bien le plus grand des éléments inférieurs à `a` dans $\{\texttt{inf}\} \cup \mathcal{E}(\texttt{abr})$.
Cas `sous_abr = Vide`. La propriété est vraie du fait que $\{\texttt{inf}\} \cup (\mathcal{E}(\texttt{abr}) \setminus \mathcal{E}(\texttt{sous\_abr})) = \{\texttt{inf}\} \cup \mathcal{E}(\texttt{abr})$.
Cas `sous_abr = Noeud ( ( e, _), abr_g, abr_d )`. On suppose la propriété vraie pour `abr_g` et `abr_d`. On notera par abus de notation `a < e`, `a = e` et `a > e` pour les différentes valeur de `comp`.

- Si `a = e`, l'algorithme renvoie `e`, on invoque les propriété **(c)** et **(d)** de `comp` et la correction est vérifiée.
- Si `a < e`, l'algorithme s'appelle récursivement sur `abr_g` et `element_inf`.
  - **(1)** On montre que `element_inf` est le plus grand des éléments inférieurs à `a` parmi $\{\texttt{inf}\} \cup (\mathcal{E}(\texttt{abr}) \setminus \mathcal{E}(\texttt{abr\_g}))$ On sait que `element_inf` est le plus grand des éléments inférieurs à `a` parmi $\{\texttt{inf}\}\cup(\mathcal{E}(\texttt{abr})\backslash\mathcal{E}(\texttt{sous\_abr}))$, or `e > a` et par définition d'un arbre binaire de recherche et **(b)**, $\forall e' \in \mathcal{E}(\texttt{abr\_d}), \texttt{e}' > \texttt{e} > \texttt{a}$. Donc `element_inf` est le plus

grand des éléments inférieurs à `a` parmi $\{\texttt{inf}\} \cup (\mathcal{E}(\texttt{abr}) \setminus \mathcal{E}(\texttt{sous\_abr})) \cup \mathcal{E}(\texttt{abr}_\texttt{d}) \cup \{\texttt{e}\} = \{\texttt{inf}\} \cup (\mathcal{E}(\texttt{abr}) \setminus \mathcal{E}(\texttt{abr\_g}))$.

– **(2)** $\mathcal{E}(\texttt{abr\_g}) \subset \mathcal{E}(\texttt{sous\_abr})$ donc $\forall e \in \mathcal{E}(\texttt{abr\_g}), \texttt{e} > \texttt{element\_inf}$.

On peut conclure ce cas par induction.

- Si `a > e`, l'algorithme s'appelle récursivement sur `abr_d` et `e`.

  – **(1)** On montre que `e` est le plus grand des éléments inférieurs à `a` parmi $\{\texttt{inf}\} \cup (\mathcal{E}(\texttt{abr}) \setminus \mathcal{E}(\texttt{abr\_d}))$ On sait que `a > e > element_inf` par hypothèse et par définition d'un arbre binaire de recherche et **(a)**, $\forall e' \in \mathcal{E}(\texttt{abr\_g}), \texttt{a} > \texttt{e} > \texttt{e}'$. Donc `e` est le plus grand des éléments inférieurs à `a` parmi $\{\texttt{inf}\} \cup (\mathcal{E}(\texttt{abr}) \setminus \mathcal{E}(\texttt{sous\_abr})) \cup \mathcal{E}(\texttt{abr}_\texttt{g}) \cup \{\texttt{e}\} = \{\texttt{inf}\} \cup (\mathcal{E}(\texttt{abr}) \setminus \mathcal{E}(\texttt{abr\_d}))$.

  – **(2)** vient de la définition d'un arbre binaire de recherche.

  On peut conclure ce cas par induction.

La propriété est bien prouvée par induction structurelle. La propriété est vraie lors de l'appel initial `parcours abr inf` si les prérequis de l'algorithme sont vérifiés. La correction est prouvée.
Complexité : L'algorithme descend l'arbre `abr` et réalise une comparaison par nœud traversé. La complexité est en $\mathcal{O}(h)$ où h est la hauteur de `abr`.

Recherche dans un Arbre Bicolore

```
let recherche_arn
    ( comp : 'a -> 'b -> ordre )
    ( a : 'a ) ( inf : 'b )
    ( arn : 'b arn ) : 'b =
  let comp_arn ( a : 'a ) ( e : 'b * couleur  ) =
    let ( b, _ ) = e in comp a b in
  let b, _ = recherche_abr comp_arn a ( inf, NOIR ) arn in b ;;
```

Complexité : La complexité est en $\mathcal{O}(\log(n))$, d'après la question précédente et puisque la hauteur d'un arbre bicolore est inférieure à $2\log(\frac{n}{2})$ (c.f. cours)

3 Recherche dans une partition

Génération aléatoire d'ensembles

Pour dénombrer le nombre d'arêtes, on parcours chaque sommet et dénombre les arêtes sortantes uniquement.

```
let nombre_aretes ( s : sommet ensemble ) : int =
  let rec parcours ( i : int ) ( nombre : int ) : int =
    if i = Array.length s then nombre
    else parcours ( i + 1 ) ( nombre + ( List.length s.(i).sortantes ) ) in
  parcours 0 0 ;;
```

Localisation dans une partition

Une *partition* est un ensemble trié de *dalles*. On utilise le type `ensemble` fourni par l'énoncé. Une dalle est une bande verticale du plan, coupée par des segments. L'énoncé propose de sauvegarder l'ensemble de ces segments dans un arbre bicolore. Ils seront triés dans l'ordre d'apparition dans la dalle, du bas vers le haut.

```
type dalle = { bornes : int * int ; segments : arete arn } ;;
type partition = dalle ensemble ;;
```

On implémente la recherche d'un point dans une partition, qui consiste à chercher la dalle dans laquelle se situe ce point, puis de chercher le segment traversant la dalle le plus haut, en dessous du point. Une fois ce segment trouvé, comme donné dans l'énoncé, la zone dans laquelle se situe le point et la zone située à gauche du segment trouvé.
On réutilise les fonctions de recherche implémentées en introduction.

```
(* Paramètre = p : un point du plan *)
(* Paramètre = partition : una partition du plan *)
(* Renvoie l'identifiant de la zone du plan dans laquelle se trouvel le point *)
let localisation_de_point ( point : point ) ( partition : partition ) : int =
  let dalle = recherche_dichotomique
      ( fun point dalle -> ( ordre_lex point dalle.bornes ) )
      point
      { bornes = (0,0) ; segments = Vide }
      partition in
  let segment_limite = recherche_arn ordre_point_arete point
      { segment = ( (0, 0), ( 2 * 524287, 0) ) ;
        zone_droite = 0 ;
        zone_gauche = 0 }
      dalle.segments in
  segment_limite.zone_gauche ;;
```

Complexité Temporelle : La recherche dichotomique est en $\mathcal{O}(\log(n_d))$ où n_d est le nombre de *dalles*. La recherche dans l'arbre bicolore est en $\mathcal{O}(\log(n_a))$ où n_a est le nombre de segments traversant la dalle. n_d est majoré par le nombre de sommets et donc n_a et n_d sont majorés tous deux pas le nombre d'arêtes n. La recherche est en $\mathcal{O}(\log(n))$.

4 Premier Algorithme de création d'une partition

Pour savoir si un segment (p, q) traverse une dalle entre x et x' il suffit de vérifier que le point p est situé « avant ou sur » la ligne verticale passant par x et que le point q est situé « après ou sur » la ligne verticale passant par x'. Pour faire cela on compare les abscisses de p et q avec x et x' respectivement.

```
(* Paramètres = segment : un segment ( un couple de points ) *)
(* Paramètres = born_inf : abscisse de la ligne horizontale de gauche *)
(* Paramètres = born_sup : abscisse de la ligne horizontale de droite *)
(* Renvoie vrai si le segment traverse la dalle *)
let traverse
    ( segment : segment ) ( borne_inf : int ) ( borne_sup : int )  : bool =
  let ( px, _ ), ( qx, _ ) = segment in
  px <= borne_inf && qx >= borne_sup ;;
```

Pour créer une dalle (de manière naïve) on parcourt toutes les arêtes du partitionnement et on insère une à une chaque arête qui traverse la dalle. Pour parcourir toutes les arêtes.

```
(* Paramètres = borne_inf : abscisse de la ligne horizontale de gauche *)
(* Paramètres = borne_sup : abscisse de la ligne horizontale de droite *)
(* Paramètres = sommets : l'ensemble des sommets et arêtes dans lequel
   la dalle est construite *)
(* Renvoie la dalle du plan délimitée par borne_inf et borne_sup *)
let creation_dalle
    ( borne_inf : int ) ( borne_sup : int )
    ( sommets : sommet ensemble ) : dalle =
  (* Paramètres = isommet : l'indice du sommet dont on ajoute les arêtes
  sortantes *)
  (* Paramètres = arete_a_ajouter : les aretes sortantes restant à ajouter *)
  (* Paramètres = arete_arn : l'arbre bicolore contenant toutes les arêtes
  traversant la dalle que l'on a rencontré à cette étape du parcours. *)
  (* Renvoie l'arbre bicolore contenant toutes les arêtes traversant la dalle
  délimitée par borne_inf et borne_sup *)
  let rec ajout_aretes
      ( isommet : int )
      ( arete_a_ajouter : arete list )
      ( arete_arn : arete arn ) : arete arn =
    match arete_a_ajouter with
    (* Tous les sommets ont été parcouru *)
    | [] when isommet = ( Array.length sommets ) - 1 -> arete_arn
    (* Toutes les arêtes sortantes du sommet ont été parcourues *)
    | [] -> ajout_aretes ( isommet + 1 ) sommets.( isommet + 1).sortantes arete_arn
    (* On croise une arete qui traverse la dalle *)
    | arete :: liste_arete when traverse arete.segment borne_inf borne_sup ->
      ajout_aretes isommet liste_arete
        ( insertion_arn ordre_lex_aretes arete arete_arn )
    (* On croise une arete qui ne traverse pas la dalle *)
    | _ :: liste_arete -> ajout_aretes isommet liste_arete arete_arn in
  { bornes = ( borne_inf, borne_sup ) ;
    segments = ajout_aretes (-1) [] Vide } ;;
```

Pour créer une partition on parcourt les sommets, triés dans l'ordre lexicographique et on lance la

création d'une dalle dès que l'on a identifiés deux sommets dont les abscisses sont consécutives dans la liste (triée) des abscisses des sommets du partitionnement.

```
(* Paramètres = sommets : l'ensemble des sommets et arêtes dans lequel
   la dalle est construite. L'ensemble doit contenir au moins un élément. *)
(* Renvoie la partition du plan *)
let creation_partition ( sommets : sommet ensemble )
  : partition =
  (* Paramètres = isommet : l'indice du sommet examiné *)
  (* Paramètres = borne_inf : l'abscisse de la ligne horizontale délimitant la
     gauche de la dalle entrain d'être calculée *)
  (* Paramètres = liste_dalles : la liste des dalles (triée de gauche à droite
     sur le plan) déjà calculées à cette étape du parcours. *)
  (* Renvoie la partition du plan *)
  let rec parcours_dalle
      ( isommet : int )
      ( borne_inf : int )
      ( liste_dalles : dalle list )
    : partition =
    if isommet = ( Array.length sommets )
    then Array.of_list ( List.rev liste_dalles )
    else begin
      let x, _ = sommets.( isommet ).point in
      if x != borne_inf then
        let dalle = ( creation_dalle borne_inf x sommets ) in
        parcours_dalle ( isommet + 1 ) x ( dalle  :: liste_dalles )
      else parcours_dalle ( isommet + 1 ) borne_inf liste_dalles
    end in
  let borne_inf, _ =  sommets.(0).point in
  parcours_dalle 1 borne_inf [] ;;
```

Complexité Temporelle : Soit n le nombre de segments du partitionnement du plan. La fonction `traverse` est en $\mathcal{O}(1)$. La fonction `creation_dalle` appelle traverse sur chaque segment et fait une insertion pour chaque arête qui traverse la dalle. Au pire des cas toutes les arêtes traverses la dalle. L'insertion est en $\mathcal{O}(\log(n))$ donc la fonction est en $\mathcal{O}(n\log(n))$. Enfin la fonction `creation_partition` appelle `creation_dalle` pour chaque dalle. Au pire des cas il y a autant de dalles que de sommets moins un. Le nombre de sommets est au pire égal au nombre d'arêtes moins un. La complexité finale est en $\mathcal{O}(n^2\log(n))$.
Complexité Spatiale : Soit n le nombre de segments du partitionnement du plan. L'espace utilisé par cet algorithme est exactement l'espace utilisé par la partition renvoyée en résultat. La fonction `creation_partition` crée un tableau de dalle. Comme argumenté ci-dessus, il y a $\mathcal{O}(n)$ dalles. Chaque dalle contient un arbre bicolore sauvegardant chaque arête traversant la dalle. Au pire, chaque arête traverse toutes les dalles. Dans ce cas l'espace utilisé par la partition est $\mathcal{O}(n^2)$.

5 Algorithme de création d'une partition à l'aide d'ARN persistants

L'algorithme crée les dalles dans l'ordre de la gauche du plan vers la droite. Un dalle est créée à partir des segments traversant la dalle précédente en traitant chaque sommet appartenant à la dalle comme expliqué dans l'énoncé. Les sommets étant triés par ordre lexicographique, tous les sommets appartenant à une dalle sont consécutifs dans l'ensemble de sommet. Le premier sommet rencontré n'étant pas dans la dalle est le premier sommet de la dalle suivante.
La fonction `creation_dalle_cole`, qui s'occupe du traitement de chaque sommet de la dalle pour créer l'arbre bicolore des segments traversant la dalle, prend soin de renvoyer le premier sommet rencontré en dehors de la dalle pour initier la création de la dalle suivante.

```
(* Paramètre = aretes_arn : arbre bicolore contenant des aretes *)
(* Paramètre = arete : la liste des arêtes à supprimer de aretes_arn *)
(* Renvoie un arbre bicolore où toutes la liste des arêtes passées en
paramètre a été supprimée de aretes_arn *)
let rec suppression_entrantes
    ( aretes_arn : arete arn ) ( aretes : arete list ) : arete arn =
match aretes with
| [] -> aretes_arn
| arete :: liste_arete ->
    suppression_entrantes
    ( suppression_arn ordre_lex_aretes arete aretes_arn ) liste_arete ;;
```

```
(* Paramètre = aretes_arn : arbre bicolore contenant des aretes *)
(* Paramètre = arete : la liste des arêtes à insérer dans aretes_arn *)
(* Renvoie un arbre bicolore où toutes la liste des arêtes passées en
    paramètre a été insérée dans aretes_arn *)
let rec insertion_sortantes
    ( aretes_arn : arete arn ) ( aretes : arete list ) : arete arn =
  match aretes with
  | [] -> aretes_arn
  | arete :: liste_arete ->
    insertion_sortantes
      ( insertion_arn ordre_lex_aretes arete aretes_arn ) liste_arete ;;
```

```
(* Paramètre = zones : l'ensemble des sommets et arêtes partitionnant le plan *)
(* Paramètre = isommet : l'indice du plus petit sommet inclu dans la dalle à
   construire *)
(* Paramètre = segments_dalle_precedente : l'arbre bicolore contenant les
   segments traversant la dalle précédente *)
(* Renvoie la dalle contenant isommet_base et l'indice du plus petit sommet de
   zones appartenant à la dalle suivante *)
let creation_dalle_persistent
    ( zones : sommet ensemble )
    ( isommet_base : int )
    ( segments_dalle_precedente : arete arn )
  : dalle * int =
  let x_inf, _ = zones.(isommet_base).point in
  (* Paramètre = isommet : l'indice du sommet en cours de traitement *)
  (* Paramètre = arete_arn : l'arbre bicolore contenant les segments traversant
     la dalle après traitement des sommets avant isommet *)
  (* Renvoie la dalle contenant isommet_base et l'indice du plus petit sommet
     de zones appartenant à la dalle suivante *)
  let rec traitement_sommet
      ( isommet : int ) ( arete_arn : arete arn ) : dalle * int =
    (* Cas de la dernière dalle, vers l'infini *)
    if isommet = ( Array.length zones )
    then ( { bornes = ( 0, 0 ) ; segments = Vide }, isommet )
    else let x, _ = zones.(isommet).point in
      (* Cas du traitement d'un sommet qui n'est plus dans la dalle de base *)
      if x != x_inf
      then ( { bornes = ( x_inf, x ) ; segments = arete_arn }, isommet )
      else traitement_sommet ( isommet + 1 )
          ( insertion_sortantes
              ( suppression_entrantes arete_arn zones.(isommet).entrantes  )
              zones.(isommet).sortantes ) in
  traitement_sommet isommet_base segments_dalle_precedente ;;
```

La fonction `creation_partition_cole` ne sert qu'à lancer successivement la fonction précédente pour créer successivement chaque dalle et les sauvegarder *in fine* dans un tableau, dans l'ordre.

```
(* Paramètre = zone : l'ensembles des sommets et arêtes partitionnant le plan *)
(* Renvoie une partition du plan *)
let creation_partition_persistent ( zones : sommet ensemble )
  : partition =
  let rec parcours_dalle
      ( isommet : int )
      ( liste_dalles : dalle list )
    : partition =
    let segments_dalle_precedente = match liste_dalles with
      | [] ->  Vide
      | dalle :: liste_dalle -> dalle.segments in
    let dalle, isommet_suivant =
      creation_dalle_persistent zones isommet segments_dalle_precedente in
    (* Fin du parcours des sommets *)
    if isommet_suivant = ( Array.length zones )
    then Array.of_list ( List.rev liste_dalles )
    (* Création de la dalle suivante *)
    else parcours_dalle isommet_suivant  ( dalle :: liste_dalles ) in
  parcours_dalle 0 [] ;;
```

Complexité Temporelle : Soit n le nombre de segments du partitionnement du plan. La fonction `creation_dalle_cole` appelle la fonction d'insertion pour chaque arête sortante de chaque sommet de la dalle et la fonction de suppression pour chaque arête entrante de chaque sommet de la dalle. Notons s_d et e_d le nombre d'arêtes sortantes et entrante, respectivement, des sommets appartenant à la dalle d. La complexité de `creation_dalle_cole` est donc $\mathcal{O}(s_d \log(n) + e_d \log(n))$. La fonction `creation_partition_cole` appelle `creation_dalle_cole` appelle `creation_dalle_cole` sur chaque dalle. L'opération est, en cumulée, en

$$\mathcal{O}(\left[\sum_{d \text{ une dalle de } \texttt{zones}} s_d\right] \log(n) + \left[\sum_{d \text{ une dalle de } \texttt{zones}} e_d\right] \log(n)) = \mathcal{O}(2\,n\,\log(n)) = \mathcal{O}(n\,\log(n))$$

La conversion de la liste en tableau et son inversion est en $\mathcal{O}(n)$ donc la complexité finale est en $\mathcal{O}(n\,\log(n))$.
Complexité Spatiale : Soit n le nombre de segments du partitionnement du plan. L'espace utilisé par cet algorithme est exactement l'espace utilisé par la partition renvoyée en résultat. La fonction `creation_partition_cole` crée un tableau de dalle construite par insertion et suppression sur la dalle précédente. Or l'insertion et la suppression ne copient pas entièrement l'arbre sur lequel elles ont travaillé, comme remarqué par l'énoncé. Le nombre de nœuds ajoutés est logarithmique en la taille de l'arbre initial. Chaque arête est insérée et supprimée une fois seulement. La première dalle a une taille en $\mathcal{O}(n)$ (si toutes les arêtes croisent cette dalle). Les insertion et suppression rajoutent en cumulée chacune $\mathcal{O}(n\,\log(n))$ nœuds au pire. La taille de la partition finale est en $\mathcal{O}(n\,\log(n))$.

Ouverture sur la partition de Sarnak-Tarjan

La recherche consiste en une recherche dans l'arbre qui, pour chaque nœud, cherche dans l'arbre droit ou gauche, à quel sous-arbre voyager selon la dalle donnée en argument. La recherche de la dalle est une recherche dans un ABR comme réalisée dans la première section.

```
type 'a gros_arn = Vide | Noeud of 'a * ( ( int * ( 'a gros_arn ) ) arn ) * ( ( int * ( 'a gr
type partition_ST = ( int * ( arete gros_arn ) arn )

let recherche_gros_arn
    ( comp : 'a -> 'b -> ordre )
    ( a : 'a ) ( i : int )
    ( inf : 'b )
    ( gros_arn : 'b gros_arn ) : 'b =
  let rec parcours ( gros_arn : 'b gros_arn ) ( element_inf : 'b ) : 'b =
    match gros_arn with
    | Vide -> element_inf
    | Noeud ( e, arn_g, arn_d ) ->
      match comp a e with
      | PLUS_PETIT ->
        let _, fils_gauche =
          recherche_arn ( fun i (j,_) -> ordre_int i j ) i ( 0, Vide ) arn_g in
        parcours fils_gauche element_inf
      | PLUS_GRAND ->
        let _, fils_droit =
          recherche_arn ( fun i (j,_) -> ordre_int i j ) i ( 0, Vide ) arn_d in
        parcours fils_droit element_inf
      | EGAUX -> e in
  parcours gros_arn inf ;;
```

6 Affichage des résultats de la fiche réponse

```
Printf.printf "Question 1 : a) %i  b) %i  c) %i  \n"
  ( u 100 ) ( u 200 ) ( u 300 ) ;;

let e212 = e 212 ;;
let e512 = e 512 ;;
let e1000 = e 1000 ;;

tri_ensemble e212 ;;
tri_ensemble e512 ;;
tri_ensemble e1000 ;;

let x1, y1 = recherche_e_n e212 121212 in
let x2, y2 = recherche_e_n e512 212121 in
let x3, y3 = recherche_e_n e1000 111222 in
Printf.printf "Question 4 : a) ( %i, %i )  b) ( %i, %i )  c) ( %i, %i )  \n"
  x1 y1 x2 y2 x3 y3 ;;

let part212 = delaunay_triangulation e212 ;;
let part512 = delaunay_triangulation e512 ;;
let part1000 = delaunay_triangulation e1000 ;;

let nb_arete212 = nombre_aretes part212 in
let nb_arete512 = nombre_aretes part512 in
let nb_arete1000 = nombre_aretes part1000 in
Printf.printf "Question 7 : a) %i  b) %i  c) %i  \n"
  nb_arete212 nb_arete512 nb_arete1000 ;;

let zone1 = localisation_de_point ( 212121, 122121 ) ( creation_partition part212 ) in
let zone2 = localisation_de_point ( 221111, 144211 ) ( creation_partition part512 ) in
let zone3 = localisation_de_point ( 111111, 212121 ) ( creation_partition part1000 ) in
Printf.printf "Question 13 : a) %i  b) %i  c) %i \n" zone1 zone2 zone3;;
```