Correction du concours blanc

Option informatique, première année

Julien Reichert

Exercice 1

```
let pente (x1, y1) (x2, y2) = if x2 = x1 then failwith "vertical" else (y2 -. y1) /. (x2 -. x1);;
(* Attention à mettre les points, l'énoncé a dit qu'on travaillait sur des flottants.
Du reste, la suite de la question laissait aussi deviner qu'on n'allait pas s'amuser
à convertir des entiers en flottants pour calculer l'arc tangente.
Noter que 1. /. 0. ne déclenche pas d'erreur
(je viens de le découvrir car j'avais un bug dans mon 3e test d'angle. *)

let pisur2 = acos 0.;;

let angle p1 p2 = try
  let ang1 = atan (pente p1 p2) in
    if fst p2 > fst p1 then ang1
    else if ang1 > 0. then ang1 -. 2. *. pisur2
    else ang1 +. 2. *. pisur2
  with _ -> pisur2 *. signe (snd p2 -. snd p1);;
(* Ne pas oublier les parenthèses ni de rattraper la division par 0.
On acceptera les valeurs approchées de pi, et on tolèrera les tentatives d'invocation d'une variable. *)
```

Exercice 2

```
(* Fonction auxiliaire : fonction affine associée à une droite non verticale. *)
let equation ((x1, y1), (x2, y2)) x = (pente (x1, y1) (x2, y2)) *. (x -. x1) +. y1;;

let position (x, y) ((x1, y1), (x2, y2)) =
  if x1 = x2 then
    if x < x1 then 1 else if x > x1 then -1 else 0
  else let f = equation ((x1, y1), (x2, y2)) in (* fonction partielle, magnifique, n'est-ce pas ? *)
    let fx = f x in if y > fx then 1 else if y < fx then -1 else 0;;
```

Exercice 3

```
(* On va tout de suite écrire deux fonctions de calcul de distance une fois pour toutes. *)
let distancecarre (x1, y1) (x2, y2) = (y2 -. y1) ** 2. +. (x2 -. x1) ** 2.;;
let distance p1 p2 = sqrt (distancecarre p1 p2);;

let hitbox_cc (p1, r1) (p2, r2) = distance p1 p2 <= r1 +. r2;;
(* Pour que les cercles s'intersectent plutôt que les disques, il faut aussi que la distance
soit supérieure à la valeur absolue de la différence des rayons. *)
```

Exercice 4

```
let hitbox_rr ((x1, y1), (x2, y2)) ((x3, y3), (x4, y4)) =
  x1 <= x4 && x3 <= x2 && y1 <= y4 && y3 <= y2;;
(* Ceci est une astuce pour déterminer si deux segments se chevauchent sur une droite,
utilisée en abscisses et en ordonnée, ce qui constitue une condition nécessaire et suffisante. *)
```

Exercice 5

```
(* Première fonction auxiliaire : deux segments parallèles sont-ils sur des droites confondues
et s'y intersectent-ils ? *)
let confondus ((x1, y1), (x2, y2)) ((x3, y3), (x4, y4)) =
  try let pente12 = pente (x1, y1) (x2, y2) in
    x1 +. pente12 *. (x3 -. x1) = y3 (* (x3, y3) est sur la droite passant par (x1, y1) et (x2, y2) *)
    && min x1 x2 <= max x3 x4 && min x3 x4 <= max x1 x2 (* astuce de l'exercice 3 *)
  with _ -> x1 = x3 && min y1 y2 <= max y3 y4 && min y3 y4 <= max y1 y2;; (* segments verticaux *)

(* Deuxième fonction auxiliaire : abscisse de l'intersection des droites prolongeant les deux segments
avec garantie d'existence et d'unicité si cette fonction est appelée. *)
let intersection_droite ((x1, y1), (x2, y2)) ((x3, y3), (x4, y4)) =
  let sitoutfoire = ref x1 in
  try
    let pente1 = pente (x1, y1) (x2, y2) in
      sitoutfoire := x3;
      let pente2 = pente (x3, y3) (x4, y4) in
        (* à ce stade, on cherche x tel que (x, y) vérifie à la fois y = y1 +. pente1 *. (x -. x1)
           et y = y3 +. pente2 *. (x -. x3), donc brouillon, substitution et réponse immédiate *)
        (y3 -. y1 -. pente2 *. x3 +. pente1 *. x1) /. (pente1 -. pente2)
  with _ -> !sitoutfoire;;
(* si une droite est verticale, il suffit de savoir laquelle et l'abscisse est triviale *)

let intersection ((x1, y1), (x2, y2)) ((x3, y3), (x4, y4)) =
  let pente1 = (try Some (pente (x1, y1) (x2, y2)) with _ -> None)
(* grosse astuce pour économiser du code *)
  and pente2 = (try Some (pente (x3, y3) (x4, y4)) with _ -> None) in
(* if dans la suite pour simplifier la compréhension et introduire des variables *)
    if pente1 = pente2
      then confondus ((x1, y1), (x2, y2)) ((x3, y3), (x4, y4))
    else let x12 = min x1 x2 and x21 = max x1 x2 and x34 = min x3 x4 and x43 = max x3 x4
      and x = intersection_droite ((x1, y1), (x2, y2)) ((x3, y3), (x4, y4)) in
        x12 <= x && x <= x21 && x34 <= x && x <= x43;;
```

Exercice 6

```
let extreme_gauche l =
  let rec coco (mx, my) ll = match ll with
     | [] -> (mx, my)
     | ((x, y)::q) -> if x < mx || x = mx && y > my
       then coco (x, y) q
       else coco (mx, my) q
  in coco (List.hd l) (List.tl l);; (* Erreur si vide *)
```

```
let positionbis (x, y) ((x1, y1), (x2, y2)) = (* -1 : à gauche du vecteur, 0 : dessus, 1 : à droite *)
  if x1 = x2 then
    if x = x1 then 0 else if (x < x1) = (y1 < y2) then -1 else 1
  else let f = equation ((x1, y1), (x2, y2)) in
    let fx = f x in if y = fx then 0 else if (y < fx) = (x1 < x2) then 1 else -1;;

let prochain (x2, y2) l =
  let rec next (mx, my) mdist ll = match ll with (* arguments redondants pour alléger *)
    | [] -> (mx, my)
    | ((x, y)::q) -> let pos = positionbis (x, y) ((x2, y2), (mx, my))
      and dist = distancecarre (x2, y2) (x, y) in
      if pos = -1 || pos = 0 && dist > mdist then next (x, y) dist q
      else next (mx, my) mdist q
  in let pt = List.hd l in
    next pt (distancecarre (x2, y2) pt) (List.tl l);;

let remove elt l = (* l sans elt et ordre chamboulé *)
  let rec aux accu ll = match ll with
  | [] -> failwith "Impossible"
  | a::q -> if a = elt then accu@q else aux (a::accu) q
  in aux [] l;;

let jarvis points =
  if List.length points < 3 then failwith "Trivial"; (* on peut rattraper cela si on veut *)
  let premier = extreme_gauche points in
  let rec boucle actuel enveloppe dispo =
    let dispobis = if actuel = premier then dispo else premier::dispo in
      let suivant = prochain actuel dispobis in
        if suivant = premier then enveloppe
        else boucle suivant (suivant::enveloppe) (remove suivant dispo)
  in boucle premier [premier] points;;
```

Exercice 7

Une façon de faire est de noter à chaque étape l'angle (en sens horaire pour simplifier) entre la verticale vers le haut et le vecteur reliant le point initial au point considéré ; cet angle augmente (il y a de la transitivité dans l'air) strictement (en raison de la restriction imposée) jusqu'à éventuellement atteindre un demi-tour, et donc être aligné verticalement avec le point initial, ce qui occasionnera une dernière étape, ou atteindre directement le point initial depuis celui qui procurait l'angle maximal.

Pour cette raison, un sommet n'est jamais revisité, même si on ne le retire pas, et la boucle conditionnelle / la pile d'appels récursifs termine avec pour variant le nombre de sommets non encore visités (classe, non ?), soit ici la taille de la liste `dispo`.

Exercice 8

La complexité est quadratique dans le pire des cas, et elle est précisément de l'ordre du produit du nombre de points au total et du nombre de points de l'enveloppe convexe (nombre de tours de boucle / d'appels récursifs).

En effet, prochain et remove sont de complexité linéaire en la taille de la liste en argument et on a vu dans la preuve de terminaison qu'il y a au plus un tour de boucle / appel récursif par élément de la liste, sachant qu'on ajoute à chaque fois un élément à l'enveloppe convexe.

Exercice 9

```
let comp (x1, y1) (x2, y2) =
  if x1 < x2 || x1 = x2 && y1 > y2 then -1
  else if x1 = x2 && y1 = y2 then 0
  else 1;;

let scinde l = let n = List.length l in
  let rec aux g d i = match i with
    | 0 -> List.rev g, d
    | _ -> aux ((List.hd d)::g) (List.tl d) (i-1)
  in aux [] l (n/2);;

let position2 (x, y) ((x1, y1), (x2, y2)) =
(* cas où deux points sont verticalement alignés entre la gauche et la droite,
il faut alors que le suivant soit à gauche ou à droite suivant le cas *)
  if x1 = x2 then
    if x > (* ! *) x1 then 1 else if x < (* ! *) x1 then -1 else 0
  else let f = equation ((x1, y1), (x2, y2)) in
    let fx = f x in if y > fx then 1 else if y < fx then -1 else 0;;

(* On commence par changer l'ordre des éléments de gauche,
car il faut la parcourir dans le sens inverse à partir d'un point d'abscisse maximale. *)

let plafond gauche droite =
  let rec ontourne accu g = match g with (* on ne peut pas se permettre de trier ! *)
    | [] -> accu
    | a::q -> if comp (List.hd accu) a = 1 then accu@(List.rev g)
      else ontourne (a::accu) q
  in let gauche2 = ontourne [List.hd gauche] (List.tl gauche) in
  let rec teste resteg rested = match resteg, rested with
    | [], _ -> failwith "Impossible"
    | _, [] -> failwith "Impossible"
    | [courantg], [courantd] -> (courantg, courantd)
    | [courantg], courantd::suivantd::q -> if position2 suivantd (courantd, courantg) = -1
      then courantg, courantd
      else teste resteg (suivantd::q)
    | courantg::suivantg::q, [courantd] -> if position suivantg (courantg, courantd) = -1
      then courantg, courantd
      else teste (suivantg::q) rested
    | courantg::suivantg::qg, courantd::suivantd::qd ->
        if position2 suivantd (courantd, courantg) = -1
          then if position suivantg (courantg, courantd) = -1 then courantg, courantd
          else teste (suivantg::qg) rested
        else teste resteg (suivantd::qd)
  in teste gauche2 droite;;

(* Il y a tout de même des difficultés supplémentaires dans la fonction plancher,
car l'ordre des listes est différent pour une bonne gestion :
gauche démarre au point le plus à droite dans le bon sens, donc il faut renverser l'accumulateur,
et droite démarre dans l'autre sens, donc il faut renverser sa queue.

En outre, les fonctions position et position2 sont échangées et les valeurs testées passent de -1 à 1.
En pratique, une bonne utilisation de positionbis aurait été meilleure,
par exemple en retournant aussi des valeurs spéciales pour les cas verticaux. *)
```

```
let plancher gauche droite =
  let rec ontourne2 accu g = match g with
    | [] -> (List.hd accu)::(List.rev (List.tl accu))
    | a::q -> if comp (List.hd accu) a = 1 then ((List.hd accu)::g)@(List.rev (List.tl accu))
      else ontourne2 (a::accu) q
  in let gauche2 = ontourne2 [List.hd gauche] (List.tl gauche) and ptdroite = List.hd droite in
  let droite2 = ptdroite::(List.rev (List.tl droite)) in
  let rec teste resteg rested = match resteg, rested with
    | [], _ -> failwith "Impossible"
    | _, [] -> failwith "Impossible"
    | [courantg], [courantd] -> (courantg, courantd)
    | [courantg], courantd::suivantd::q -> if position suivantd (courantd, courantg) = 1
      then courantg, courantd
      else teste resteg (suivantd::q)
    | courantg::suivantg::q, [courantd] -> if position2 suivantg (courantg, courantd) = 1
      then courantg, courantd
      else teste (suivantg::q) rested
    | courantg::suivantg::qg, courantd::suivantd::qd ->
        if position suivantd (courantd, courantg) = 1
          then if position2 suivantg (courantg, courantd) = 1 then courantg, courantd
          else teste (suivantg::qg) rested
        else teste resteg (suivantd::qd)
  in teste gauche2 droite2;;

let cxcv gauche droite =
  let (ghaut, dhaut) = plafond gauche droite and (gbas, dbas) = plancher gauche droite in
    let rec parcours accu phase g d = match phase with
(*
  phase 0 : parcourir gauche jusqu'au plafond,
  phase 1 : sauter à droite et dérouler jusqu'à trouver où on est arrivé,
  phase 2 : parcourir droite jusqu'au plancher,
  phase 3 : sauter à gauche et dérouler jusqu'à trouver où on est arrivé,
  phase 4 : parcourir gauche jusqu'à la fin
*)
      | 0 ->
        (match g with
          | [] -> failwith "Impossible"
          | (a::q) -> parcours (a::accu) (if a = ghaut then 1 else 0) q d)
      | 1 ->
        (match d with
          | [] -> failwith "Impossible"
          | (a::q) -> parcours accu (if a = dhaut then 2 else 1) g (if a = dhaut then a::q else q))
      | 2 ->
        (match d with
          | [] -> failwith "Impossible"
          | (a::q) -> parcours (a::accu) (if a = dbas then 3 else 2) g q)
      | 3 ->
        (match g with
          | [] -> List.rev accu (* cas où gbas est le point de départ, qui n'a pas été repris dans g *)
          | (a::q) -> parcours accu (if a = gbas then 4 else 3) (if a = gbas then a::q else q) d)
      | _ -> (List.rev accu)@g (* gain de temps, pas besoin de filtrer *)
    in parcours [] 0 gauche droite;;

(* On observe qu'il faut parenthéser les filtrages intérieurs,
en raison de l'ambiguïté de l'enchaînement de deux filtrages,
que Caml lève en donnant la priorité au dernier match rencontré. *)
```

```
let enveloppe_base points = match points with
| [] -> []
| [a] -> points
| [a; b] -> List.sort comp points (* le plus à gauche et en haut en tête... *)
| _ -> match List.sort comp points with (* pas besoin de parenthèses en raison des priorités *)
  | [a; b; c] -> let p = positionbis c (a, b) in
    if p = 0 then [a; c] (* on a 3 points alignés, donc l'enveloppe relie les extrémaux *)
    else if p = -1 then [a; c; b]
    else [a; b; c] (* ... et parcours en sens horaire *)
  | _ -> failwith "Trop d'éléments";;

let enveloppe points =
  let pts = List.sort comp points in
    let rec enveloppe_dpr l =
      if List.length l < 4 then enveloppe_base l (* ou jarvis si les petits cas sont gérés *)
      else let g, d = scinde l in cxcv (enveloppe_dpr g) (enveloppe_dpr d)
    in enveloppe_dpr pts;;
```

Exercice 10

On dispose de cas de base (ici, des ensembles de deux ou trois points) et la taille des ensembles est divisée par deux à chaque appel récursif imbriqué, en particulier elle décroit strictement et peut servir de variant pour prouver la terminaison au niveau des appels principaux.

Concernant la recherche du plafond et du plancher, on sait que la position de chaque point par rapport aux droites étudiées successivement doit changer à un moment, car les droites seraient toujours plus hautes sinon (strictement sauf cas de points alignés, nécessairement au nombre de deux vu la gestion des alignements dans le programme). Comme on finit par retomber sur le point de départ, on a un point sous la dernière droite au pire des cas à ce moment.

Exercice 11

Le Master theorem s'invoque avec $c_n = 2c_{n/2} + O(n)$ car on appelle la fonction récursive sur les deux moitiés et on a un coût linéaire pour séparer en deux ainsi que pour chercher le plafond et le plancher et pour rassembler le tout. Ceci donne un $O(n \log n)$ qui s'additionne au coût du tri, identique pour les bons tris.