Mpsi 1 et 2 **Devoir surveillé 3**

Les algorithmes seront écrits en code CAML et seront accompagnés de commentaires et d'explications qui permettront de les comprendre aisément.

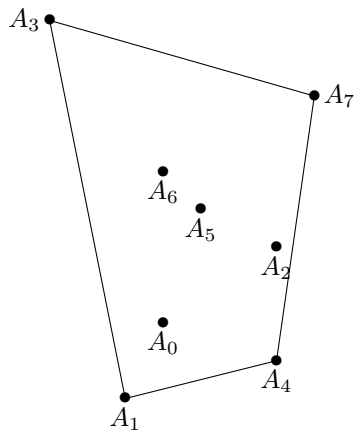
Problème 1 - Enveloppe convexe

Une partie E du plan géométrique est dite convexe si et seulement si pour tout $(A, B) \in E^2$, le segment $[AB]$ est inclus dans E. Par exemple, l'intérieur d'un triangle, un disque, l'intérieur d'un carré sont des parties convexes, alors que l'intérieur d'un croissant ne l'est pas.

On vérifie sans peine (ce n'est pas demandé) que l'intersection d'une famille de parties convexes est une partie convexe.

Alors si E est une partie quelconque du plan, on forme l'intersection de toutes les parties convexes qui contiennent E : c'est la plus petite partie convexe qui contient E, on l'appelle l'enveloppe convexe de E, noté $\text{conv}(E)$.

On considère un ensemble fini de points du plan $E = \{A_0, \dots, A_{n-1}\}$. L'objectif est de construire algorithmiquement l'enveloppe convexe de E. Il est facile de montrer que dans ce cas, $\text{conv}(E)$ est l'intérieur d'un polygone convexe, dont les sommets sont pris parmi les points de E, comme dans la figure suivante. Par abus, on conviendra de dire qu'un ensemble fini E est convexe pour signifier que son enveloppe convexe a pour sommets tous les points de E (avec cet abus de langage, on dira qu'un triangle est convexe).



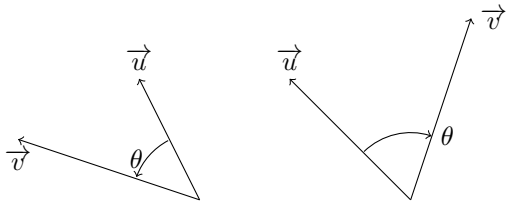
On représente les points du plan par des couples de deux flottants (a, b) de type `float * float`, qu'on notera `p` (« point » de CAML) : il représente un point du plan noté P. On rappelle que la première composante d'un tel couple `p` est calculable par la fonction `fst p`, et la seconde par `snd p`. Pour simplifier, on suppose que les points n'auront jamais d'abscisses égales, ni d'ordonnées égales.

On représente les ensembles finis de points par des listes de tels couples ; leurs enveloppes convexes étant parfaitement connues par la donnée de quelques sommets, on les représente aussi par des listes, auxquelles on ajoute une contrainte : les points doivent être donnés dans l'ordre trigonométrique et le premier point de la liste est celui le plus haut.

Dans l'exemple précédent, si on note $E = \{A_0, \dots, A_7\}$, alors E est représenté par la liste `[a0; ... ; a7]` et son enveloppe convexe par la liste `[a3; a1; a4; a7]`.

Enfin, on définit la notion d'orientation dans le plan. Soit $\overrightarrow{u}$, $\overrightarrow{v}$ deux vecteurs non nuls du plan. On note θ l'angle orienté du vecteur $\overrightarrow{u}$ vers $\overrightarrow{v}$, dont on choisit la mesure dans $]-\pi, +\pi]$.

On dit qu'on passe de $\overrightarrow{u}$ à $\overrightarrow{v}$ par une rotation gauche (resp. droite) si et seulement si $\theta \in [0, \pi]$ (resp $[-\pi, 0]$) autrement dit si on passe de $\overrightarrow{u}$ à $\overrightarrow{v}$ en tournant dans le sens trigonométrique (resp. dans le sens rétrograde).



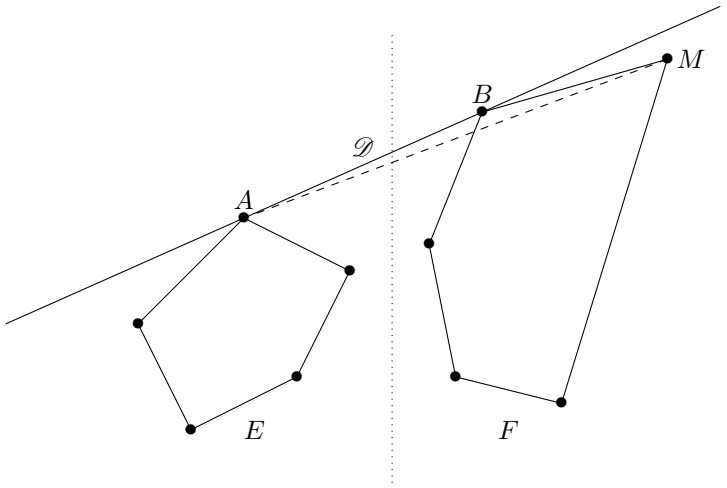
rotation gauche rotation droite

Si on connaît les coordonnées de $\overrightarrow{u}$ et $\overrightarrow{v}$ dans une base orthonormée directe, alors on peut déterminer facilement le sens de la rotation de $\overrightarrow{u}$ vers $\overrightarrow{v}$: il est donné par le signe de $\sin\theta$. Si $\sin\theta \geqslant 0$, alors on passe de $\overrightarrow{u}$ vers $\overrightarrow{v}$ par une rotation gauche, sinon par une rotation droite. Or si le calcul de θ est difficile, celui du signe de $\sin\theta$ est aisé.

On montre en effet que $\det(\overrightarrow{u}, \overrightarrow{v}) = ||\overrightarrow{u}|| \times ||\overrightarrow{v}|| \times \sin\theta$ (ici le déterminant est calculé dans la base orthonormée directe du plan). Le signe de $\det(\overrightarrow{u}, \overrightarrow{v})$ donne donc le sens de la rotation de $\overrightarrow{u}$ vers $\overrightarrow{v}$.

Partie 1 - Plus haute et plus basse tangentes communes à deux convexes

Soit E et F deux ensembles finis non vide de points, qu'on suppose convexes et séparés par une droite verticale : E est situé à gauche de la droite et F à droite. On appelle tangente commune à E et à F toute droite $\mathscr{D}$ qui passe par un sommet de E et par un sommet de F telle que E et F soient contenus dans le même demi-plan délimité par $\mathscr{D}$, comme dans la figure suivante.



Pour trouver la tangente haute, on détermine le point M le plus haut de F et on fait varier un point dans E, jusqu'à trouver la droite la plus haute possible, c'est-à-dire celle qui est obtenue en tournant le plus possible à droite (en tirets sur le dessin) ; elle passe par le point A de E. Puis on fixe ce point A et on fait varier un point sur F, jusqu'à obtenir la droite la plus haute possible, cette fois-ci en tournant le plus possible à gauche : elle passe par B. La droite (AB) est alors une tangente commune telle que E et F soient en-dessous.

Question 1)

a) Écrivez une fonction `vecteur p q` de type `point -> point -> float * float`, qui calcule les coordonnées du vecteur $\overrightarrow{PQ}$.

b) Écrivez une fonction `determinant a b c` de type `point -> point -> point -> float`, qui calcule le déterminant du couple de vecteurs $(\overrightarrow{AB}, \overrightarrow{AC})$.

c) Écrivez enfin une fonction `agauche a b c`, de type `point -> point -> point -> bool`, qui détermine si on passe de $\overrightarrow{AB}$ à $\overrightarrow{AC}$ par rotation gauche.

Question 2) Écrivez une fonction `tangente_gauche li p`, de type `point list -> point -> point` qui prend en paramètres une liste de points `li`, un point `p` situé à droite de tous les points de la liste et qui retourne le point `q` de la liste tel que la droite (PQ) soit la plus haute possible.

Question 3) Écrivez une fonction `tangente_droite li p`, de type `point list -> point -> point` qui prend en paramètres une liste de points `li`, un point `p` situé à gauche de tous les points de la liste et qui retourne le point `q` de la liste tel que la droite (PQ) soit la plus haute possible.

Question 4) Écrivez une fonction `tangente_haute e f`, de type `point list -> point list -> (point * point)` qui calcule la tangente obtenue comme dans l'exemple.

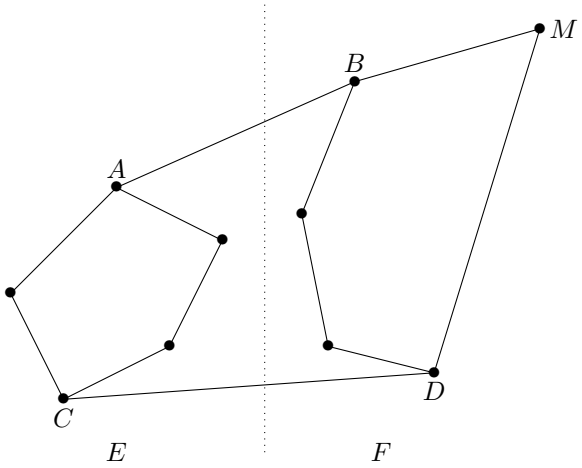
Question 5) Écrivez de même une ou plusieurs fonctions nécessaires au calcul d'une tangente basse située sous les deux convexes.

Question 6) En fonction des longueurs des listes, donnez les complexités de ces fonctions.

Partie 2 - Réunion de deux convexes

Soit E et F deux ensembles finis non vide de points, qu'on suppose convexes et séparés par une droite verticale, E à gauche et F à droite. On suppose avoir calculé 4 points A, B, C, D tels que A, C soient dans E et B, D dans F et tels que la droite (AB) soit une tangente haute aux deux convexes E et F et la droite (CD) soit une tangente basse.

On peut fabriquer un nouvel ensemble convexe à partir de E et F : on part du point A, on suit les points dans l'ordre trigonométrique dans E jusqu'à arriver sur C, on passe à D et on suit les points dans l'ordre trigonométrique dans F jusqu'à arriver sur le point B. La liste de ces points ne respecte pas tout à fait la contrainte, mais par une simple permutation circulaire, on peut s'arranger pour la rendre correcte.



Si E et F sont donnés par des listes de points e et f ayant les contraintes précisées plus haut, alors il faut retrouver les bons points des listes e et f, qu'on pourrait appeler « points entre `a` et `c` dans l'ordre trigonométrique » d'une part, « points entre `d` et `b` dans l'ordre trigonométrique » d'autre part. Par exemple, pour la première liste, ces points sont vraiment entre `a` et `c` si `a` est avant `c` dans la liste ou ils sont au contraire à l'extérieur de `a` et `c` dans le cas contraire :

si `e` = [... ; `a`; ..... `c`; ...] , alors ces points sont [.....]

si `e` = [..... ; `c`; ... ; `a`; .. ..] , alors ces points sont [.. .. ; .....]

Question 1) Écrivez une fonction `apres a li` de type `'a -> 'a list -> 'a list`, qui calcule la liste des éléments qui sont situés après (au sens strict) `a` dans la liste `li` (on suppose que `a` est bien un élément de `li` : il est donc inutile de perdre son temps à traiter des cas d'erreurs).

Question 2) Écrivez une fonction `avant a li` de type `'a -> 'a list -> 'a list`, qui calcule la liste des éléments qui sont situés avant (au sens strict) `a` dans la liste `li` (même remarque).

Question 3) Écrivez une fonction `est_avant a c li` , de type `'a -> 'a -> 'a list -> bool` qui dit si `a` est avant `c` dans la liste `li` (on suppose encore que `a` et `c` sont dans la liste).

Question 4) Écrivez une fonction `entre a c e` de type `point -> point -> point list -> point list`, qui calcule la liste des éléments de `e` qui sont entre `a` et `c` quand on tourne dans le sens trigonométrique.

Question 5) Écrivez une fonction `reunion e f` de type `point list -> point list -> point list`, qui calcule la liste des points de l'enveloppe convexe de la réunion $E \cup F$. On pourra remarquer que le point le plus haut de e et de f appartient forcément à cette enveloppe convexe.

Question 6) Donnez les complexités de ces fonctions en fonction de la longueur des listes.

Partie 3 - Calcul de l'enveloppe convexe

Soit E un ensemble fini non vide de points du plan. Si son cardinal est petit, il est facile de calculer son enveloppe convexe. Sinon on le coupe en deux sous-ensembles de même cardinal (à un près) par une droite verticale...

Question 1) Expliquez comment on peut déterminer une telle droite à partir de la liste `e` représentant E et à quel coût, qu'on voudra pas trop grand !

Question 2) Terminez la description de l'algorithme et donnez sa complexité en fonction du nombre de points de E.

Problème 2 - Encore des permutations

Soit $n \in \mathbb{N}^*$, on appelle tableau-permutation d'ordre n tout tableau de longueur n dont les n cases contiennent les entiers $0, 1, \ldots, n-1$ dans un certain ordre.

Un tel tableau représente aussi une bijection φ de $[\![0, n-1]\!]$ dans $[\![0, n-1]\!]$: l'application $\varphi : i \mapsto t.(i)$

Question 1) Si φ est la bijection associée à un tableau-permutation t, écrivez une fonction qui construit le tableau associé à la réciproque φ^{-1}.

Question 2) Si φ est la bijection associée à un tableau-permutation t, écrivez une fonction qui construit l'orbite d'un entier x sous l'action de φ, c'est-à-dire la suite finie $(x, \varphi(x), \varphi \circ \varphi(x), \ldots)$: vous choisirez une représentation de cet ensemble et vous justifierez votre choix.

Question 3) Si φ est la bijection associée à un tableau-permutation t, écrivez une fonction qui calcule la décomposition de φ en cycles disjoints. Là encore vous choisirez une représentation informatique adaptée. Quelle est la complexité de votre fonction (en fonction du paramètre n) ?

Question 4) Écrivez une fonction qui calcule la signature d'une telle permutation.

Problème 1

Partie 1

Question 1)

a)

```
let vecteur p q =
  (fst q -. fst p, snd q -. snd p);;
```

b)

```
let determinant a b c =
  let u = vecteur a b and v = vecteur a c in
  fst u *. snd v -. snd u *. fst v;;
```

c)

```
let agauche a b c =
 determinant a b c >= 0.;;
```

Question 2) On applique le principe suivant : si on connaît déjà un point candidat q et qu'on teste un autre point r, on le choisit si le vecteur $\overrightarrow{PR}$ est obtenue par rotation droite du vecteur $\overrightarrow{PQ}$.

```
let rec tangente_gauche li p =
  match li with
  | [r] -> r
  | r :: qu -> let q = tangente_gauche qu p in
    if not(agauche p q r) then r else q;;
```

Question 3) La même fonction que la précédente, en échangeant droite et gauche, rotation gauche et rotation droite.

```
let rec tangente_droite li p =
  match li with
  | [r] -> r
  | r :: qu -> let q = tangente_droite qu p in
    if agauche p q r then r else q;;
```

Question 4)

```
let tangente_haute e f =
 let m = hd f in
 let a = tangente_gauche e m in
 let b = tangente_droite f a in
 (a, b);;
```

Question 5)

```
let sym (x,y) = (-. x,y);;

let rec symetrise li =
 match li with
 | [] -> []
 | p :: qu -> sym p :: symetrise qu;;

let rec plus_haut li =
 match li with
 | [p] -> p
 | q :: qu -> let p = plus_haut qu in
    if snd q > snd p then q else p;;

let rec tangente_basse e f =
 let e' = symetrise e and f' = symetrise f in
```

```
 let m' = plus_haut f' in
 let a' = tangente_gauche e' m' in
 let b' = tangente_droite f' a' in
 (sym a', sym b');;
```

On symétrise la figure par rapport à l'axe (Ox) : la tangente basse entre e et f est alors la symétrique de la tangente haute entre e' et f'.

Question 6) Il est clair que toutes les fonction précédentes parcourent les listes une seule fois et qu'à chaque appel récursif, les opérations sont élémentaires, donc les complexités vérifient la relation de récurrence $C(n) = C(n-1) + O(1)$ donc elles sont toutes en $O(n)$: les algorithmes précédents sont de complexité linéaire.

La fonction tangente_haute fait un appel à deux fonctions de complexité linéaire par rapport aux longueurs des deux listes, donc sa complexité est en $O(n_1 + n_2)$.

Partie 2

Question 1)

```
let rec apres a li =
 match li with
 | x :: qu -> if x = a then q else apres a qu;;
```

Question 2)

```
let rec avant a li =
 match li with
 | x :: qu -> if x = a then [] else x :: avant a qu;;
```

Question 3) Si la liste commence par a, alors a est avant c; si la liste commence par c, alors a est aprés c; sinon, on enlève la tête : a et c sont dans la queue et leur position relative n'a pas changé.

```
let rec est_avant a c li =
 match li with
 | x :: qu -> if x = a then true
   else if x = c then false
   else est_avant a c qu;;
```

Question 4) Si a est avant c dans e, alors les éléments qui sont entre a et c dans le sens trigonométrique sont vraiment entre a et c. Sinon, ces éléments sont ceux qui sont après a suivis de ceux qui sont avant c.

```
let entre a c e =
 if est_avant a c e then
   avant c (apres a e)
 else
   apres a e @ avant c e;;
```

Question 5)

```
let reunion e f =
  let a,b = tangente_haute e f in
  let c,d = tangente_basse e f in
  let l = (b :: a :: (entre a c e)) @ (c :: d :: entre d b f) in
  let m = plus_haut (hd e) (hd f) in
  (m :: apres m l) @ avant m l;;
```

Question 6) Pareil que dans la partie 1 , sachant que l'opérateur `@` a une complexité linéaire en la longueur de la première liste.

Partie 3

Question 1) Il suffit de trier la liste selon les abscisses, ce qui peut se faire en $O(n \log n)$, puis de sélectionner un élément central (en $O(n)$) en calculant d'abord la longueur de la liste (en $O(n)$). Au total, on peut séparer la liste initiale en $O(n \log n)$.

Question 2) On a vu dans les parties précédentes qu'on pouvait fusionner deux convexes en un seul au prix d'une complexité linéaire. Et on vient de voir qu'on pouvait diviser une liste en $O(n \log n)$ en deux listes de points... C'est un algorithme qui utilise la méthode diviser pour régner.

Mais le th. du cours ne s'applique pas directement à cause du coût du tri en $O(n \log n)$... L'astuce consiste à trier une bonne fois pour toute au début avant le début de l'algorithme récursif et lorsqu'on divise la liste, à conserver l'ordre des éléments ! Ainsi on ne trie qu'une seule fois avant de débuter l'algorithme proprement dit et donc toutes les divisions se feront en $O(n)$ (coût de la sélection d'un objet dans une liste, le calcul de la longueur n'étant même plus nécessaire puisqu'on la connaît à chaque appel récursif).

Conclusion : le coût de l'algorithme récursif suit une récurrence du type $C(n) = 2C(n/2) + O(n)$, donc $C(n) = O(n \log n)$ (th. du cours). On ajoute encore le coût d'un tri en $O(n \log n)$, ce qui donne un coût total du même ordre de grandeur pour le calcul de l'enveloppe convexe.

Problème 2

Question 1)

```
let inverse t =
  let n = vect_length t in
  let u = make_vect n 0 in
  for i = 0 to (n-1) do
    u.(t.(i)) <- i
  done;
  u;;
```

Question 2)

```
let orbite x t =
  let rec orbaux y l =
    let z = t.(y) in
    if z = x then rev l
    else orbaux z (z :: l)
  in
  orbaux x [x];;
```

Question 3)

```
let noninvar t =
  let n = vect_length t in
  let r = ref 0 in
  while !r < n && t.(!r) = !r do
    incr r (* ou r := !r + 1 *)
  done;
  if !r = n then -1 else !r;;

let rec modif t l =
  match l with
  | [] -> ()
  | x ::  q -> t.(x) <- x; modif t q;;

let cycles t =
  let rec cycl lc =
    let r = noninvar t in
    if r = -1 then lc
    else
       let x = r in
       let orb = orbite x t in
       modif t orb;
       let lc' = orb :: lc in
       cycl lc'
  in
  cycl [];;
```

La fonction noninvar t calcule l'indice i dans le tableau t du premier élément tel que $t.(i) \neq i$, s'il existe ; et si t représente l'application identité, la fonction calcule -1.

La fonction modif t l parcourt la liste ℓ et pour chaque entier i de cette liste, change la valeur de $t.(i)$ en i. Cette fonction sert en fait à « effacer » les éléments d'un cycle qu'on vient de calculer, pour éviter de le recalculer par la suite.

La dernière fonction calcule la liste des cycles, qui sont en fait les orbites des éléments non invariants du tableau.

La fonction noninvar est de complexité linéaire par rapport à la longueur du tableau : on parcourt le tableau de longueur n une fois et à chaque étape, les opérations sont élémentaires.

Pareil pour la fonction modif, de complexité linéaire par rapport à la longueur de la liste.

La dernière fonction est au pire de complexité quadratique : il y a au plus n cycles (en fait, au plus $n/2$) et pour chaque cycle, on utilise la fonction noninvar sur le tableau t (coût $O(n)$), la fonction orbite ($O(n)$) et la fonction modif ($O(n)$), donc la complexité est au plus $n \times O(n) = O(n^2)$.

Si on est plus attentif, on peut dire mieux : si c est le nombre de cycles, et si le plus long cycle est de longueur p, alors le coût est au plus $c \times O(n + 2p)$, sachant que $c \times p \leqslant n$. Le coût principal est issu de la fonction noninvar : si on modifie celle-ci pour recommencer la recherche du premier non invariant **à partir de la position du premier non invariant précédent**, on ne parcourt plus que le tableau une seule fois. Le coût peut être ramené à $O(n) + c \times O(p) = O(n)$, car $c \times p \leqslant n$.

Question 4) Application directe du cours de mathématiques : la signature est un morphisme pour la composition des applications vers la multiplication des nombres et la signature d'un cycle est (-1) à la puissance sa longueur moins un.

```
let puiss n =
  if n mod 2 = 0 then 1 else -1;

let sign t =
  let lc = cycles t in
  let rec prod lc =
    match lc with
    | [] -> 1
    | l :: qc -> puiss (list_length l - 1) * prod qc
  in
  prod lc;;
```