DS1 MPSI : Enveloppe convexe

Rappelons que le temps d'exécution d'un programme A (fonction ou procédure) est le nombre d'opérations élémentaires (comparaisons, additions, soustractions, multiplications, divisions, affectations, etc.) nécessaires à l'exécution de A. Sauf mention contraire dans l'énoncé du sujet, l.a.e étudiant.e n'aura pas à justifier des temps de calcul de ses programmes. Toutefois, iel devra veiller à ce que ces derniers ne dépassent pas les bornes prescrites.

Un *nuage de points* est ici un ensemble de points du plan. Pour simplifier ces points ont des coordonnées entières. Un nuage de points est représenté par une liste de listes de deux entiers `int list list`. Par exemple, le nuage de la figure 1 est implémenté par le code suivant :

Listing 1 – Un nuage de points

```
(*une matrice 12 lignes de 2 colonnes*)
let nuage = [[0;0];[1;4];[1;8];[4;1];[4;4];[5;9];
             [5;6];[7;-1];[7;2];[8;5];[11;6];[13;1]];;
```

Ce sujet a pour objectif de calculer des enveloppes convexes de nuages de points dans le plan affine, un grand classique en géométrie algorithmique. On rappelle qu'un ensemble $C \subset \mathbb{R}^2$ est convexe si et seulement si pour toute paire de points $p, q \in C$, le segment de droite $[p, q]$ est inclus dans C. L'enveloppe convexe d'un ensemble $P \subset \mathbb{R}^2$, notée $\text{Conv}(P)$, est le plus petit convexe contenant P . Dans le cas où P est un ensemble fini (appelé nuage de points), le bord de $\text{Conv}(P)$ est un polygône convexe dont les sommets appartiennent à P.

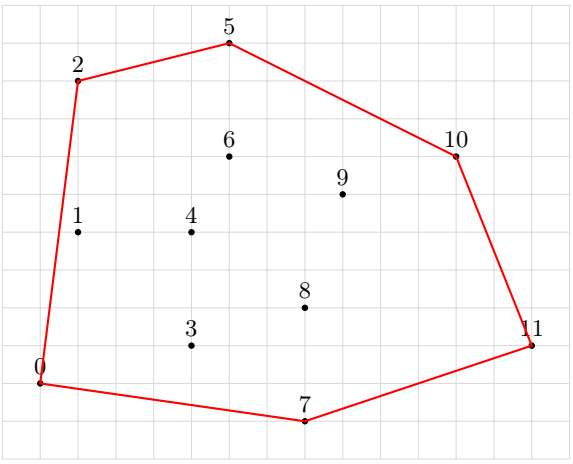

FIGURE 1 – Nuage de points et enveloppe convexe

Le calcul de l'enveloppe convexe d'un nuage de points est un problème fondamental en informatique, qui trouve des applications dans de nombreux domaines comme :

— la robotique, par exemple pour l'accélération de la détection de collisions dans le cadre de la planification de trajectoire,

— le traitement d'images et la vision, par exemple pour la détection d'objets convexes (comme des plaques minéralogiques de voiture) dans des scènes 2d,

— l'informatique graphique, par exemple pour l'accélération du rendu de scènes 3d par lancer de rayons,

— la théorie des jeux, par exemple pour déterminer l'existence d'équilibres de Nash,

— la vérification formelle, par exemple pour déterminer si une variable risque de dépasser sa capacité de stockage ou d'atteindre un ensemble de valeurs interdites lors de l'exécution d'une boucle dans un programme, et bien d'autres encore.

Dans ce sujet nous allons implémenter deux algorithmes de calcul du bord de l'enveloppe convexe d'un nuage de points P dans le plan affine. L'*algorithme du paquet cadeau* part du point le plus bas du nuage de points P et consiste à envelopper le nuage progressivement en faisant pivoter une droite tout autour. L'*algorithme de balayage* part du point le plus à gauche du nuage et gère deux piles représentant les parties supérieures et inférieures de l'enveloppe convexe en les mettant à jour itérativement.

Dans toute la suite on supposera que le nuage de points P est de taille $n \geq 3$ et *en position générale*, c'est-à-dire qu'il ne contient pas 3 points distincts alignés.

Ces hypothèses vont permettre de simplifier les calculs en ignorant les cas pathologiques, comme par exemple la présence de 3 points alignés sur le bord de l'enveloppe convexe. Nos programmes prendront en entrée un nuage de points P dont les coordonnées sont stockées dans une matrice à deux colonnes.

Dans cette partie, les nuages passés en argument sont des listes de listes de deux entiers et, chaque liste de deux entiers représentant un point. Il n'y a pas 3 points alignés dans le nuage (en particulier le nuage est sans doublon). **On ne DEMANDE PAS de vérifier ces propriétés** dans les fonctions qui suivent.

Fonctions outils

Q.1 Écrire la fonction `rev : 'a list -> 'a list` qui inverse le contenu d'une liste. L'appel `rev[1;2;3]` produit par exemple `[3;2;1]`.

Q.2 Écrire la fonction `depack : int list -> int * int` qui prend en paramètre un point 2D (une liste de 2 entiers) et renvoie ses deux coordonnées.

```
# depack [1;2];;
- : int * int = (1, 2)
# depack [1;2;3];;
Exception: Failure "pas une liste de 2 valeurs".
```

Q.3 Écrire la fonction `plusbas : int list list -> int list` qui prend en paramètre un nuage de points et renvoie celui qui a la plus petite ordonnée. Si plusieurs points sont possibles, la fonction renvoie le premier d'entre eux.

```
# plusbas nuage;;
- : int list = [7; -1]
```

Nous aurons besoin dans la suite d'effectuer un seul type de test géométrique : celui de l'*orientation*.

Définition 1. Étant donnés trois points p_i, p_j, p_k du nuage P , distincts ou non, le test d'orientation renvoie $+1$ si la séquence (p_i, p_j, p_k) est orientée positivement, -1 si elle est orientée négativement, et 0 si les trois points sont alignés

Remarque. Si le nuage vérifie bien l'hypothèse de position générale et si l'orientation du triplet d epoints est nulle cela signifie que deux des 3 points sont égaux.

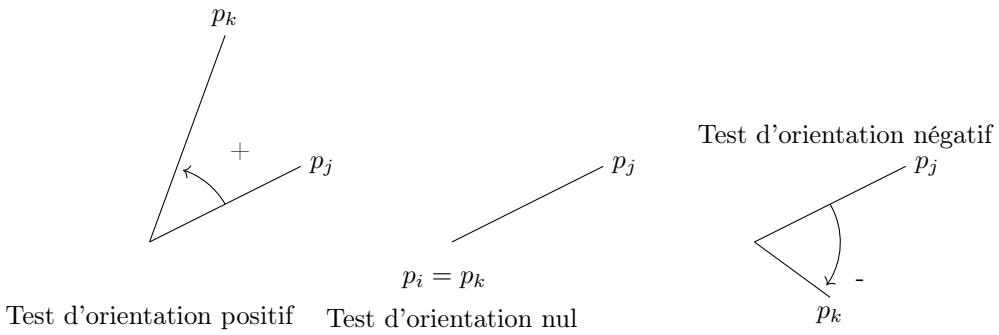

FIGURE 2 – Test d'orientations

Voir l'exemple figure 2.

Pour déterminer l'orientation de (p_i, p_j, p_k), il suffit de calculer l'aire signée du triangle, comme illustré sur la figure ci-dessous. Cette aire est la moitié du déterminant de la matrice 2×2 formée par les coordonnées des vecteurs $\overrightarrow{p_i p_j}$,$\overrightarrow{p_i p_k}$.

Rappel. Le *déterminant* de deux vecteurs du plan $\vec{u} = (x, y), \vec{v} = (x', y')$ est la quantité notée $\begin{vmatrix} x & x' \\ y & y' \end{vmatrix}$ et valant $xy' - yx'$.

Q.4 Écrire la fonction `oriente : int list -> int list -> int list -> int` qui prend en paramètres 3 points p_0, p_1, p_2 et renvoie le résultat (-1, 0 ou +1) du test d'orientation sur la séquence (p_i, p_j, p_k).

Algorithme du papier cadeau

Cet algorithme a été proposé par R. Jarvis en 1973. Il consiste à envelopper peu à peu le nuage de points P dans une sorte de paquet cadeau, qui à la fin du processus est exactement le bord de $\mathrm{Conv}(P)$. On commence par insérer le point de plus petite ordonnée (le point $(7, -1)$ dans l'exemple de la figure 1) dans le paquet cadeau, puis à chaque étape de la procédure on sélectionne le prochain point du nuage P à insérer.

La procédure de sélection fonctionne comme suit. Soit p_i le dernier point inséré dans le paquet cadeau à cet instant. Par exemple, $(11, 6)$ dans l'exemple de la figure 3. Considérons la relation binaire $\preccurlyeq$ définie sur l'ensemble $P \setminus \{pi\}$ par : $pj \preccurlyeq pk \iff \mathrm{oriente}(p_i, p_j, p_k) \leq 0$ (l'angle de vecteur $\overrightarrow{p_i p_j}, \overrightarrow{p_i p_k}$ a une mesure dans $[-\pi, 0]$).

La relation $\preccurlyeq$ est une relation d'ordre total. Le prochain point à insérer (le point d'indice 5 dans la figure 3) est l'élément maximum pour la relation d'ordre $\preccurlyeq$. Il peut se calculer en temps linéaire en fonction de la taille du nuage en effectuant une simple itération sur les points de $P \setminus \{p_i\}$.

Q.5 Justifier brièvement le fait que $\preccurlyeq$ est une relation d'ordre sur l'ensemble $P \setminus \{p_i\}$, c'est-à-dire :

- (réflexivité) pour tout $j \neq i, p_j \preccurlyeq p_j$;
- (antisymétrie) pour tous $j, k \neq i, p_j \preccurlyeq p_k$ et $p_k \preccurlyeq pj$ implique $j = k$;
- (transitivité) pour tous $j, k, l \neq i, p_j \preccurlyeq p_k$ et $p_k \preccurlyeq p_l$ implique $p_j \preccurlyeq p_l$;
- (totalité) pour tous $j, k \neq i, p_j \preccurlyeq p_k$ ou $p_k \preccurlyeq p_j$.

Solution. Si l'orientation est nulle les points p_i, p_j, p_k sont alignés. Compte tenu de l'hypothèse générale, cela impose $j = k$.

- Reflexivité. le déterminant de $\overrightarrow{p_i p_j}$ avec $\overrightarrow{p_i p_j}$ vaut 0.

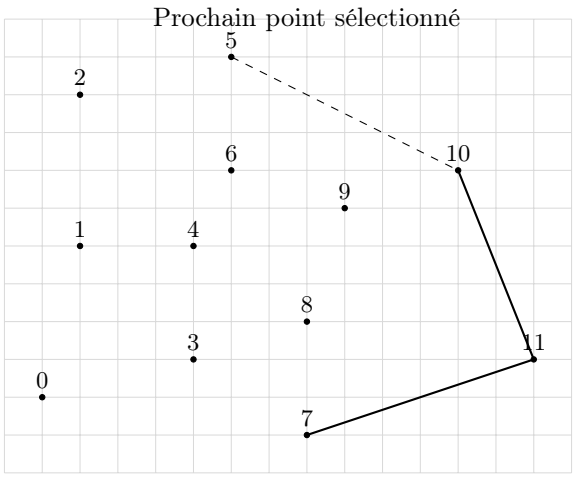

FIGURE 3 – Prochain point sélectionné si le point courant est $(11, 6)$ (10ème point du nuage)

— antisymétrie. Les deux déterminants $\det(\overrightarrow{p_ip_j}, \overrightarrow{p_ip_k})$ et $\det(\overrightarrow{p_ip_k}, \overrightarrow{p_ip_j})$ sont de signes opposés par propriété du déterminant mais ils doivent être négatifs par hypothèse. Seule possibilité : le déterminant est nul. Dans ce cas les vecteurs $\overrightarrow{p_ip_j}$ et $\overrightarrow{p_ip_k}$ sont colinéraires. Cela impose que les trois points soient alignés. Mais par hypothèse générale, il faut que deux points soients égaux. $p_k = p_j$

— transitivité. On suppose qu'aucune orientation n'est nulle, sinon il y a des points confondus. Les angles $(\overrightarrow{p_ip_k}, \overrightarrow{p_ip_j})$ et $(\overrightarrow{p_ip_l}, \overrightarrow{p_ip_k})$ sont de mesure entre 0 et π. Pour que $p_j \preccurlyeq p_l$, i faudrait que $(\overrightarrow{p_ip_l}, \overrightarrow{p_ip_j})$ soit aussi de mesure entre 0 et π. Mais la relation de chasles peut donner un angle plus grand que π. Si c'était le cas, le point i ne pourrait pas être sur l'enveloppe convexe (à affiner)

— ordre total : on peut toujours calculer le déterminant de deux vecteurs.

□

Q.6 Écrire la fonction `next : int list list -> int list -> int list` qui prend en paramètre un nuage de points et un point A de ce nuage supposé sur l'enveloppe convexe. La fonction renvoie le prochain point de l'envelope convexe tel que calculé dans la description ci-dessus.

```
# next nuage [11;6] = [5;9];;
- : bool = true
```

L'*algorithme du papier cadeau* consiste à partir d'une liste contenant uniquement le point le plus bas du nuage. Puis à insérer de proche en proche les points successifs qui réalisent le maximum de la relation d'ordre $\preceq$ correspondant au point courant. L'algorithme s'arrête lorsqu'on retombe sur le point de départ.

La terminaison et la correction de cet algorithme sont admises.

Q.7 Écrire la fonction `convJarvis : int list list -> int list list` qui réalise l'algorithme du papier cadeau et renvoie l'enveloppe convexe (sous forme de liste de points) du nuage passé en argument.

```
# convJarvis nuage=
[[7; -1]; [13; 1]; [11; 6]; [5; 9];
 [1; 8]; [0; 0]; [7; -1]];;
- : bool = true
```

Solution. Voici

Listing 2 – rev

```
let rev l =
  let rec aux acc l = match l with
    | [] -> acc
    | x::q -> aux (x::acc) q
in aux [] l;;
```

Listing 3 – depack

```
let depack l = match l with
  | [x;y]->x,y
  | _ -> failwith "pas une liste de 2 valeurs";;
```

Listing 4 – oriente

```
let oriente p0 p1 p2 =
  let x0,y0=depack p0 in
  let x1,y1=depack p1 in
  let x2,y2=depack p2 in
  let x01,y01=x1-x0,y1-y0 in
  let x02,y02=x2-x0,y2-y0 in
  let d = x01*y02-y01*x02 in
  match d with
  | 0 -> 0
  | alpha when alpha > 0 -> 1
  | _ -> -1;;
```

Listing 5 – plusbas

```
let plusbas l =
  match l with
  | [x;y]::q ->
    let rec aux q m =
      match q with
      | [] -> m
      | [x;y]::q ->
        let _,by = depack m in
        if by > y
        then aux q [x;y]
        else aux q m
      | _ -> failwith "something's wrong"
    in aux q [x;y]
  | _-> failwith "pb";;
```

Listing 6 – depack

```
let depack l = match l with
  | [x;y]->x,y
  | _ -> failwith "pas une liste de 2 valeurs";;
```

Il s'agit d'un simple algorithme de recherche de maximum.

Listing 7 – next

```
let next nuage p = (*pb si p est le 1er point du nuage !*)
  let first = match nuage with
    | p1 :: p2 :: q -> if p1 = p then p2 else p1
    | _ -> failwith "nuage de moins de 2 points"
  in let rec aux l m = (*m max provisoire*)
       match l with
       | [] -> m
       | [a;b] :: q->
         if  oriente p m [a;b] < 0
         then aux q [a;b]
         else aux q m
       | _-> failwith "pb"
  in aux nuage first;;
```

Listing 8 – convJarvis

```
  let p0 = plusbas nuage in
  let rec aux acc p =
    (*acc : env conx en construction
      p : dernier point ajouté *)
    let p' = next nuage p in
    let acc' = p'::acc in
    if p' = p0
    then acc'
    else aux acc' p'
  in List.rev @@ aux [p0] p0;;
```

A chaque itération dans `convJarvis`, on ajoute un nouveau point de l'enveloppe en faisant une recherche de maximum par un parcourt du nuage. Cette opération coûte $O(n)$.

On s'arrête dès que le maximum pour le point courant est le point de départ (le point le plus bas).

Si on admet qu'en agissant ainsi, on retrouve bien tous les points de l'enveloppe convexe (preuve de correction non demandée), alors on réalise k recherche de maximum.

La complexité est un $O(kn)$.

□

On a supposé que les points du nuage vérifiaient l'hypothèse de position générale et on ne demandait pas de tester si c'était bien le cas. Vérifions-le maintenant :

Q.8 Écrire la fonction `alignes (nuage: int list list) : bool` qui prend en paramètre un nuage de points (supposé sans doublon) et renvoie le bouléen vrai si et seulement si le nuage contient 3 points distincts alignés.

On ne demande pas de vérifier que le nuage ne contient pas de doublon.

```
# not @@ alignes nuage;;
- : bool = true
# let nuage2  = [[0;0];[1;4];[1;8];[4;1];[4;4];
                 [5;9];[10;-13];[5;6];[7;-1];[7;2];[8;5];
                 [11;6];[13;1]]
in alignes nuage2;;
- : bool = true
```

Q.9 Exprimer la complexité temporelle au pire de l'appel `convJarvis c` où `c` est un nuage de n points dont l'enveloppe convexe en comporte k.

Solution. voici le code :

Listing 9 – alignes

```
let rec alignes (nuage: int list list) : bool =
  let rec aligne2 nuage p0 p1 = match nuage with
    | [] -> false
    | p2 :: _ when p0 <> p2 && p1 <> p2 && oriente p0 p1 p2 = 0 -> true
    | _::q -> aligne2 q p0 p1
  in let rec aligne1 nuage p0 = match nuage with
      | [] -> false
      | p1 :: _ when p0 <> p1 && aligne2 nuage p0 p1 -> true
      | _ :: q -> aligne1 q p0
  in match nuage with
  | [] -> false
  | p0 :: _ when aligne1 nuage p0 -> true
  | _ :: q ->
     (*pas 2 points alignés avec le 1er point*)
     alignes q;;
```

□

Algorithme du balayage

On rappelle qu'un nuage de points est une liste de listes de deux coordonnées entières.

On dit qu'un nuage possède la *propriété de l'abscisse croissante* si les points sont rangés par première coordonnée croissante.

Q.10 Écrire la fonction `is_sorted : int list list -> bool` qui prend en paramètre un nuage et vérifie si il possède la propriété de l'abscisse croissante.

À partir de maintenant, on suppose que les nuages de points fournis en entrée possèdent la propriété d el'abscise croisante. De la sorte, le premier point du nuage a toujours la plus petite abscisse.

On ne demande pas de vérifier cette propriété dans les fonctions qui suivent.

Nous calculons maintenant l'enveloppe convexe par l'algorithme dit *de balayage* qui été proposé par R. Graham en 1972 et plus précisément sa variante (plus simple) proposée par A. Andrew quelques années plus tard. Cet algorithme utilise deux *piles*, c'est-à-dire une structure qui se comporte comme une pile d'assiette : on accède sans effort à l'assiette du dessus, mais pour récupérer l'assiette du fond, il faut dépiler une à une les assiettes au-dessus. Pas de panique : c'est exactement ainsi que se comportent les listes OCaml. Nous utiliserons donc les listes comme des piles, ainsi que nous le faisons naturellement depuis toujours. Enfin introduisons le vocabulaire suivant : un *empilement* est le fait d'ajouter un élément au sommet de la pile ; un *dépilement* consiste à retirer le sommet de la pile.

L'idée de l'algorithme de balayage est de balayer le nuage de points horizontalement de gauche à droite par une droite verticale, tout en mettant à jour l'enveloppe convexe des points de P situés à gauche de cette droite, comme illustré dans la figure 7.

Plus précisément, l'algorithme visite chaque point de P une fois, par ordre croissant d'abscisse (donc par ordre croissant d'indice de colonne dans le tableau tab car celui-ci est trié). À chaque nouveau point p_i visité, il met à jour le bord de l'enveloppe convexe du sous-nuage $\{p_0, \ldots, p_i\}$ situé à gauche de p_i.

On remarque que les points p_0 et p_i sont sur ce bord, et on appelle *enveloppe supérieure* la partie du bord de Conv $\{p_0, \ldots, p_i\}$ située au-dessus de la droite passant par p_0 et p_i (p_0 et p_i compris), et *enveloppe inférieure* la partie du bord de Conv$p_0, \ldots, p_i$ située au-dessous (p_0 et p_i compris). Le bord de Conv $\{p_0, \ldots, p_{n-1}\}$ est donc constitué de l'union de ces deux enveloppes, après suppression des doublons de p_i.

Informatiquement, les coordonnées des sommets des enveloppes inférieure et supérieure seront stockés dans deux piles de points : E_i (notée informatiquement `ei` pour enveloppe inférieure) et E_s (notée informatiquement `es` pour enveloppe supérieure).

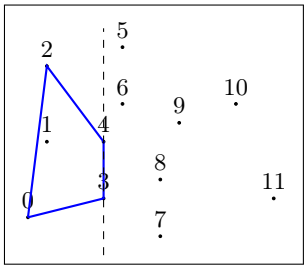

FIGURE 4 – Étape 1

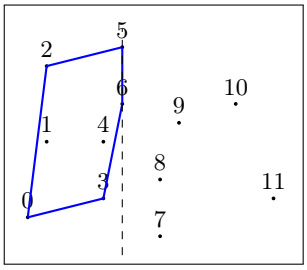

FIGURE 5 – Étape 2

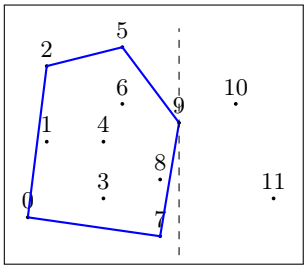

FIGURE 6 – Étape 3

FIGURE 7 – Diverses étapes dans la procédure de balayage

Par exemple, dans le cas du nuage P de la figure 7 gauche, le sous-nuage $\{p_0, p_1, p_2, p_3, p_4\}$ a pour enveloppe supérieure `es` la pile $[p_4; p_2; p_0]$ et pour enveloppe inférieure `ei` la pile $[p_4; p_3; p_0]$. On constate que le nouveau point est au sommet de la pile, le point d'abscisse minimum étant à la base de la pile. Quant à lui, le bord de son enveloppe convexe est donné par la liste $[p_0; p_3; p_4; p_2; p_0]$.

La mise à jour de l'enveloppe supérieure est illustrée dans la figure 11 : tant que le point visité (p_9 dans ce cas) et les deux points dont les indices sont situés au sommet de la pile `es` (dans l'ordre : p_8 et p_5) forme une séquence (p_9, p_8, p_5) d'orientation négative , on dépile le point situé au sommet de `es` (p_8 dans ce cas). On poursuit ce processus d'élimination jusqu'à ce que l'orientation devienne positive ou qu'il ne reste plus qu'un seul point dans la pile. Le point visité (p_9 dans ce cas) est alors inséré au sommet de `es`. La mise à jour de l'enveloppe inférieure s'opère de manière symétrique.

Q.11 Écrire la fonction `majES : int list list -> int list -> int list list` telle que `majES es p` met à jour l'enveloppe supérieure `es` en y ajoutant le nouveau point `p` et en retirant les points nécessaires.

```
# majES  [[7; 2]; [5; 9]; [1; 8]; [0; 0]] [8;5];;
- : int list list = [[8; 5]; [5; 9]; [1; 8]; [0; 0]]
```

Q.12 Écrire la fonction `majEI : int list list -> int list -> int list list` telle que `majEI ei p` met à jour l'enveloppe inférieure `ei` en y ajoutant le nouveau point `p` et en retirant les points nécessaires.

```
# majEI [[3;2];[4;0]] [6;1];;
- : int list list = [[6; 1]; [4; 0]]
```

Q.13 Écrire la fonction `deuxEnv : int list list -> int list list * int list list` qui renvoie les deux enveloppes inférieure et supérieure du nuage (dans cet ordre).

```
# nuage;;
- : int list list =
[[0; 0]; [1; 4]; [1; 8]; [4; 1]; [4; 4]; [5; 9];
[5; 6]; [7; -1]; [7; 2]; [8; 5]; [11; 6]; [13; 1]]
# deuxEnv nuage;;
- : int list list * int list list =
([[13; 1]; [7; -1]; [0; 0]],
[[13; 1]; [11; 6]; [5; 9]; [1; 8]; [0; 0]])
# deuxEnv [[1;2];[3;4]];;
Exception: Failure "il faut au moins 3 points".
```

On veut maintenant calculer la complexité de l'appel `deuxEnv cloud` pour un nuage `cloud` de n points. On rappelle que cette fonction parcourt les points du nuage et met à jour pour chaque point les deux listes `es` et `ei` qui représentent l'enveloppe supérieure et l'enveloppe inférieure.

Q.14 Le calcul de complexité utilisé ici consiste à évaluer le coût total de toutes les mises à jour des deux enveloppes. On appelle cette méthode d'évaluation de la complexité *analyse par agrégat*. On se concentre d'abord sur les mises à jour de l'enveloppe supérieure `es`, l'étude de celles de `ei` étant similaire.

(a) Combien de fois au total un même point P du nuage est-il empilé dans `es` ?

Solution. Exactement une fois □

(b) Majorer le nombre total de fois où un même point est retiré de la pile `es`.

Solution. Au plus une fois. □

(c) Majorer le nombre total d'opérations d'empilement/dépilement effectuées pour les mises à jour successives de la pile `es`.

Solution. Chaque point étant empilé une fois et dépilé au plus 1 fois, cela fait au plus 2 opérations empiler/dépiler par point donc $2n$ opérations au maximum. □

(d) Comparer le nombre total de calcul d'orientations et le nombre total d'empilements/dépilements pour les mises à jour de `es`. Justifier grossièrement.

Solution. Il y a moins de calculs d'orientations que de calculs d'empilement/dépilement.
Par exemple, si je dois retirer deux points pour empiler le nouveau point, je dois effectuer 3 calculs d'orientations (deux négatifs, un positif) et j'ai effectué un empilement et 2 dépilements.
En revanche, si la pile a 1 élément, je ne fais aucun calcul d'orientation pour empiler.
Donc le nombre total de calcul d'orientations est inférieur au nombre total d'empilements/dépilements. majoré par $2n$. □

(e) Pour les mises à jour de `es`, le coût total des filtrages est de l'ordre de grandeur du nombre d'empilements/dépilements. Le nombre total de comparaisons avec zéro est égal au nombre de calculs d'orientation.

i. Estimer en fonction de n le coût de la création et de toutes les mises à jour de `es` lors de l'appel `deuxEnv cloud` pour un nuage de taille n. On attend une réponse de la forme $O(f(n))$ avec une justification.

ii. En déduire la complexité de l'appel `deuxEnv cloud` en fonction de n.

Solution. Pour la totalité des appels à `majES` : Il y a au plus $2n$ opérations d'empilements/dépilements, idem pour les filtrages, les comparaisons avec zéro et les calculs d'orientation.
Toutes ces opérations sont en $O(1)$. Cela nous fait un coût en $O(n)$.
Idem pour `majEI`.

Pour l'appel `deuxEnv nuage`, il faut ajouter à ce qui précède les n comparaisons pour filtrage. Donc $O(n)$ opérations supplémentaires.
On obtient une complexité en $O(n)$. □

Solution. **Preuve 2**
On peut aussi étudier la complexité amortie des mises à jour. Nous comptons uniquement le nombre de calculs d'orientations.
Intéressons nous à une séquence de n mises à jours de l'enveloppe supérieure ES (en étudiant un à un les n points du nuage). Notons C_i la complexité de l'appel `majES` ES_i p_i avec ES_i le contenu

de l'ES après l'insertion du point numéro $i-1$ et p_i le point numéro i. C_i est donc le nombre de calculs d'orientation.

Comme potentiel, on pose $\varphi(ES_i) = \ell_i$ où ℓ_i est la longueur de ES_i. Ce potentiel est nul au départ puisque la liste est vide.

Le coût amorti de l'opération `majES` ES_i p_i est noté A_i et vaut $C_i + \Delta\varphi$.

$$A_i = C_i + (\ell_i - \ell_{i-1}) = C_i((\ell_{i-1} - (C_i - 1)) - \ell_{i-1}) = 1$$

Ainsi le coût amorti est constant. D'après le corollaire du th. d'amortissement, la complexité amortie de chaque mise à jour est un $O(1)$ (cela n'est pas demandé mais c'est en bonus).

Sur n opérations, la complexité réelle des MAJ de l'ES est un $O(n)$. Certes nous n'avons compté que les calculs d'orientation, mais le nombre des autres opérations à coût constants (comparaisons de filtrage, comparaison à zéro, ajout d'un élément à une liste...) est une fonction affine du nombre d'appels à orientation. Ainsi, la complexité de toutes les MAJ de ES est un $O(n)$.

Comme c'est aussi le cas pour la complexité des MAJ de l'EI, on en déduit que la complexité du calcul des deux enveloppes est un $O(n)$.

□

Il ne reste plus qu'à calculer l'enveloppe convexe.

Q.15 Écrire la fonction `convGraham : int list list -> int list list` telle que `convGraham nuage` renvoie l'enveloppe convexe du nuage sous la forme d'une liste de points dont le premier élément et le dernier sont égaux au premier point du nuage et dont tout triplet de points consécutifs possède une orientation strictement positive.

```
# convGraham nuage;;
- : int list list =
[[0; 0]; [7; -1]; [13; 1]; [11; 6]; [5; 9]; [1; 8]; [0; 0]]
```

Solution. Voici

Listing 10 – is_sorted

```
let rec is_sorted lst =
  match lst with
  | [] | [_] -> true  (* liste vide ou à un seul point *)
  | [x1; _] :: ([x2; _] :: _ as t) ->
      if x1 <= x2 then is_sorted t
      else false
  | _ -> false  (* au cas où un point n'a pas deux coordonnées *)
;;
```

Listing 11 – majES

```
let rec majES es p =
  match es with
  | [] | [_] -> p::es
  | p1::p2::q ->
     if oriente p p1 p2 >= 0 (*p1 est le premier point plus grand que son successeur*)
     then p::es
     else majES (p2::q) p;;
```

Listing 12 – majEI

```
let rec majEI ei p =
  match ei with
  | [] | [_] -> p::ei
  | p1::p2::q ->
     if oriente p p1 p2 <= 0
     then p::ei
     else majEI (p2::q) p;;

majEI [[3;2];[4;0]] [6;1];;
```

Listing 13 – deuxEnv

```
let deuxEnv nuage = match nuage with
    [] | [_] | [_;_] ->
          failwith "pas assez de pts"
    | p0::p1::q ->
       let rec aux es ei c =
         match c with
         | [] -> ei, es
         | p::q ->
            aux (majES es p) (majEI ei p)
              q
       in aux [p0] [p0] (p1::q);;
```

Listing 14 – convGraham

```
let convGraham nuage =
  let ei, es = deuxEnv nuage in
  (List.rev ei)
  @ (List.tl es);;
```

□

Appendice

Gestion des exceptions

Nous connaissons déjà les exceptions de type `Failure`. Nous les utilisons pour signaler qu'un paramètre passé en argument d'une fonction est non conforme à la sémantique.

Par exemple, la fonction suivante renvoie le premier élément d'une liste s'il existe et soulève une exception accompagnée d'un message d'erreur sinon :

```
let first l = match l with
  | [] -> failwith "liste vide"
  | x::_ -> x;;
```

```
# first [];;
Exception: Failure "liste vide".
# first [1;2;3];;
- : int = 1
```

Une exception est *bloquante* : sans traitement, le processus ne peut plus continuer son exécution. Avec la construction `try ... with ... ->`, on peut traiter l'exception et poursuivre l'exécution. La fonction suivante utilise le déclenchement de l'exception dans `first` pour indiquer si la liste passée en argument est vide ou non :

```
let empty l =
  try
    let _ = first l in false
  with Failure _ ->true;;
```

```
# empty [], empty [1;2;3];;
- : bool * bool = (true, false)
```

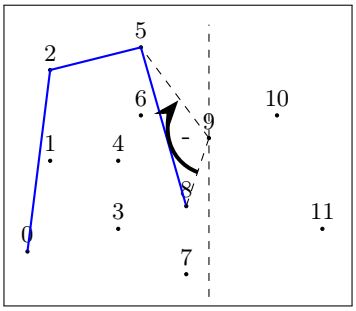

FIGURE 8 – Étape 1

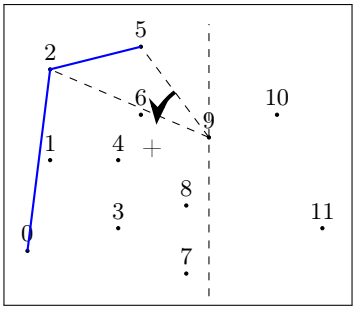

FIGURE 9 – Étape 2

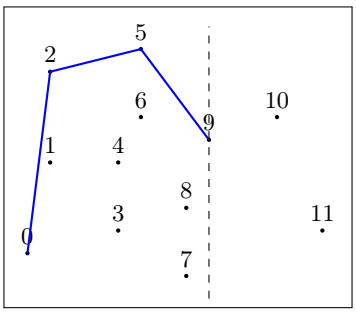

FIGURE 10 – Étape 3

FIGURE 11 – Mise à jour de l'enveloppe supérieure lors de la visite du point 9