

DS3 MP2I : Slow-Fast et Enveloppe convexe par balayage

Démonstration.

□

Avertissement Les réponses sont à écrire exclusivement sur le document réponse fourni. Le nom de l'étudiant.e doit figurer sur toutes les feuilles. Les pages doivent être rendues dans l'ordre et non écornées. Tout passage peu lisible dans une copie sera ignoré au moment de la correction même si la réponse qu'il contient est correcte. Le non respect de ces consignes sera fortement pénalisé dans la note.

- Il est toujours possible de répondre à une question en utilisant la réponse d'une question précédente même si cette dernière a été laissée de côté. L'inverse n'est pas autorisé.
- Les complexités demandées sont toujours les complexité temporelles au pire. Toutes les opérations standards sur les entiers et les booléens sont de complexité $O(1)$.
- Lorsqu'une étude de complexité est attendue, le résultat est exprimé sous la forme $O(f(n))$ où n est la *taille du problème* et f est une fonction la plus simple possible. Si la taille du problème est composé de plusieurs paramètres, par exemple n et m , on attend un résultat comme $O(f(n, m))$.
- On fait de la *science* : tout résultat (en particulier les complexités ou les conversion de formats -binaire, CA2, etc.-) doit être expliqué.
- Comme il est d'usage en CPGE, on réserve les notations droites pour les objets informatique et les notations en italique pour les expressions mathématiques. Par exemple, `x` désigne une variable en C et x désigne le même objet mais dans un contexte mathématique.
- Une mauvaise note n'évalue qu'une copie pas une personne. Un devoir noté 0 n'est pas le signe que l'étudiant concerné est nul, mais indique juste que son devoir est très mauvais ce jour là.

Dans ce devoir, les récursions terminales ne sont pas imposées.

Enfin, la seule fonction de bibliothèque autorisée est `List.rev : 'a list -> 'a list` qui inverse une liste.

1 Algorithme Slow-Fast

L'*algorithme du lièvre et de la tortue*, également connu sous le nom d'*algorithme de détection de cycle de Floyd* ou encore *Slow-Fast*, est un algorithme pour détecter un cycle dans une suite récurrente engendrée par une certaine fonction f définie d'un ensemble fini S dans lui-même.

1.1 Aspect mathématique

Les résultats de cette section peuvent être admis dans les sections suivantes.

Définition 1. Soit $(u_n)_{n \in \mathbb{N}}$. On dit que $(u_n)_{n \in \mathbb{N}}$ est *périodique* à partir d'un certain rang si il existe m et k tels que si $n \geq m$ alors $u_n = u_{n+k}$.

Dans ce cas on dit que k est *une* période de $(u_n)_{n \in \mathbb{N}}$. La plus petite période μ de (u_n) est appelée LA période de $(u_n)_{n \in \mathbb{N}}$.

On admet la propriété :

Proposition 1.1. Soit $(u_n)_{n \in \mathbb{N}}$ une suite périodique à partir d'un certain rang. Alors toute période est multiple de LA période.

Démonstration.

Remarque. Trop d'étudiants ont pris cette propriété pour une équivalence.

La propriété dit qu'une période est un multiple de LA période MAIS PAS qu'un multiple de la période est une période.

□

Soit S un ensemble fini de cardinal n et $f: S \rightarrow S$ une fonction (on dit que f est définie et *stable* sur S). On considère dans toute cette section la suite $(u_i)_{i \in \mathbb{N}}$ définie par $u_0 \in S$ et, pour tout $i \in \mathbb{N}$, $u_{i+1} = f(u_i)$.

Q.1 Montrer qu'il existe deux entiers p, k avec $k > 0$ et $u_p = u_{p+k}$.

Q.2 Montrer que la suite $(u_i)_{i \in \mathbb{N}}$ est périodique à partir de p et que k est *une* période, c'est à dire que $\forall n \geq m$, $u_{n+k} = u_n$.

Dans toute la suite on considère p et k vérifiant la propriété ci-dessus.

Q.3 Montrer qu'il existe $q \geq p$ tel que q est également une période de la suite.

Q.4 Montrer qu'il existe $m \in \mathbb{N}^*$ tel que $u_m = u_{2m}$.

Q.5 Soit m_1 le plus petit nombre tel que $m_1 > m$ et $u_{m_1} = u_{2m_1}$. Montrer que LA période de la suite $(u_i)_{i \in \mathbb{N}}$ est $\mu = m_1 - m$.

Démonstration. — Par principe des tiroirs, existent p, k tels que $u_p = u_{p+k}$.

— Soit $n > p$. Alors $u_n = f^{n-p}(u_p)$. Et $u_{n+k} = f^{n-p}(u_{p+k})$. Par transitivité, il vient $u_{n+k} = f^{n-p}(u_p) = u_n$.

— On peut choisir $k \geq p$ puisque $u_p = u_{p+k} = u_{p+k+k} = \dots$. Comme existe v tel que $vk > p$, on a $u_p = u_{p+vk}$. On pose $q = vk$

— On a, en prenant $q \geq p$ comme ci-dessus,

$$u_{p+1} = u_{p+q+1}$$

Et donc, par récurrence immédiate, pour tout r

$$u_{p+r} = u_{p+q+r}$$

On cherche r de sorte que $p + q + r = 2p + 2r$. Il vient $r = q - p \geq 0$.

Alors $u_{p+r} = u_{2(p+r)}$ et on a trouvé m .

— On a $m_1 \leq m + \mu$. En effet $u_{2m+2\mu} = u_{2m} = u_m = u_{m+\mu}$ et m_1 est le plus petit nombre plus grand que m à vérifier cette propriété.

Donc $\mu \geq m_1 - m$. Or μ divise m et m_1 . Donc μ divise aussi $m_1 - m$.

Le nombre μ divise $m_1 - m$ et il est plus grand. Donc $\mu = m_1 - m$.

Remarque. Autre preuve :

Les nombres m et m_1 sont deux périodes donc deux multiples de la période. S'il existe une période m' strictement comprise entre m et m_1 , alors $u_{m'} = u_{2m'}$ et $m < m' < m_1$: c'est absurde par minimalité de m_1 .

Ainsi, m et m' sont deux multiples consécutifs de la période.

Par conséquent, il existe k tels que $m = k\mu$ et $m_1 = (k+1)\mu$. Par suite $m - m_1 = \mu$.

Certains élèves sont allés un peu vite en besogne en posant directement $m = k\mu$ et $m_1 = (k+1)\mu$ sans justification. □

Observons que, si une suite définie par une fonction $f : S \rightarrow S$ avec S non nécessairement finie vérifie qu'il existe $p, k > 0$ vérifiant $u_p = u_{p+k}$, alors les questions 2 à 5 sont encore valables dans ce contexte.

1.2 Recherche naïve

On l'a vu, si la fonction $f : S \rightarrow S$ est définie sur un ensemble fini S , la liste `[e, f(e), f(f(e)), ...]` comporte deux éléments égaux mais de rangs distincts. On recherche ici de façon naïve deux de ces indices.

Q.6 Écrire la fonction `trouver_indice : 'a -> 'a list -> int` qui renvoie la position d'un élément cherché dans une liste et soulève une exception si cet élément est absent.

```
1 # trouver_indice 1 [10;20;30;8;1;7;9];;
2 - : int = 4
3 # trouver_indice 1 [10;20;30;8;10;7;9];;
4 Exception: Failure "pas trouvé".
```

Q.7 Donner la complexité de l'appel `trouver_indice x l` pour une liste de taille n . On ne demande pas une analyse détaillée mais on indiquera au moins l'équation de récurrence de la complexité au pire.

Démonstration. $C_n = C_n + 1 = O(n)$ □

Q.8 Écrire la fonction `doublon : 'a list -> int * int` qui renvoie le premier couple d'indices (n, m) correspondant à deux valeurs identiques dans la liste. Le premier indice doit être le plus petit possible.

```
1 # doublon [10;1;2;8;3;1;6;4;1];;
2 - : int * int = (1, 5)
3 # doublon [1;2;8;3;0;6;4];;
4 Exception: Failure "pas de doublon".
```

On pourra consulter avec profit dans l'appendice la partie réservée au traitement des exceptions.

Q.9 Donner la complexité de l'appel `doublon l` pour une liste de taille n . On ne demande pas une analyse détaillée mais on indiquera au moins l'équation de récurrence de la complexité au pire.

Démonstration. $C_{n+1} = C_n + n = O(n^2)$ □

1.3 Méthode de Floyd

Soit $f : S \rightarrow S$ une fonction d'un ensemble fini S dans lui-même. On sait que la suite (u_i) définie par $u_0 \in S$ et $u_{i+1} = f(u_i)$ est périodique. Nous voulons en déterminer la période.

Alors qu'un algorithme naïf consisterait à stocker chaque élément de la suite et, à chaque étape, de vérifier si le nouvel élément fait partie des éléments déjà rencontrés, l'algorithme de Floyd, lui, utilise un espace constant et les propriétés mathématiques vues plus haut.

Q.10 Recherche d'un couple $u_m = u_{2m}$.

- Écrire la fonction `find (f: ('a -> 'a)) (u0 : 'a) : int * 'a` qui prend en paramètre une fonction f définie et stable sur un ensemble, un élément originel u_0 et renvoie, s'ils existent, un couple (m, u_m) tel que $m > 0$ et $u_m = u_{2m}$.
- Est-ce que votre algorithme termine si f peut prendre une infinité de valeurs ?
- La situation de la question précédente peut-elle se produire sur un ordinateur actuel ?

Démonstration. Si f peut prendre un nombre non borné de valeurs, rien ne dit qu'on trouve $u_m = u_{2m}$. Boucle infinie possible.

Et non, un ordinateur actuel a une mémoire finie donc une fonction ne peut prendre qu'un nombre fini de valeurs. \square

Q.11 Écrire la fonction `periode (f: 'a->'a) (u0:'a) : int` qui prend en paramètres une fonction f stable sur un ensemble finie (on ne demande pas de vérifier ce fait) et un élément originel u_0 . L'appel `periode f u0` renvoie LA période de la suite (u_i) définie par $u_0 \in S$ et $u_{i+1} = f(u_i)$.

Remarque. Le problème de l'arrêt consiste à déterminer à l'aide d'un ordinateur théorique de mémoire INFINIE pour tout couple (A, e) où A est un algorithme et e est une entrée de A si le calcul $A(e)$ s'arrête ou non. Il est connu pour être *indécidable*.

Pour un ordinateur réel (donc à mémoire FINIE), en revanche, le problème de l'arrêt est décidable. Comme la mémoire contenant à la fois les instructions et les données sur lesquelles elles s'appliquent, il vient qu'à un état mémoire donné ne peut succéder qu'un unique autre état mémoire parfaitement déterminé (les ordinateurs sont bien modélisés par des machines de Turing). Pour détecter une boucle infinie, il suffit donc d'établir que le processus passe par deux états mémoires identiques. Un programme *superviseur* lance simultanément deux processus correspondants à un même programme et applique l'algorithme Slow-Fast. Il scanne itérativement la mémoire après chaque instruction pour le premier processus et après chaque paire d'instructions pour le second puis compare les deux. Soit le second processus (le plus rapide) termine (et le problème de la terminaison est alors résolu) soit l'algorithme Slow-Fast détecte une égalité des états mémoire et donc une boucle infinie. En pratique cependant, la taille de la mémoire réservée aux processus rend illusoire la découverte d'une boucle infinie dans un temps raisonnable sauf si on accorde peu de mémoire aux processus pour s'exécuter.

2 Enveloppe convexe

Cette partie fait suite au DM4 rendu précédemment.

Dans ce sujet, un *point* du plan est représenté par une liste de deux entiers : son abscisse et son ordonnée (les points ont ici des coordonnées entières).

Un *nuage de points* est une liste de points (voir 1).

Listing 1 – Un nuage de points

```

1 || let nuage = [[0;0];[1;4];[1;8];[4;1];[4;4];[5;9];
2 ||   [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 par la *méthode de balayage*. 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 .

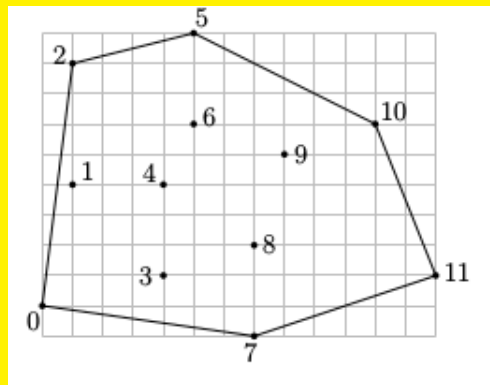


FIGURE 1 – Un nuage de points et son enveloppe convexe (les points sont numérotés ici de 0 à 11 par abscisse croissante)

Pour des raisons de place, on n'indique dans les schémas que les numéros des points du nuage et non leur nom complet.

2.1 Tri des données

L'*algorithme de balayage* présenté à la sous-section suivante travaille à partir d'un nuage de points rangés par abscisses croissantes. Nous avons donc besoin d'un tri.

Q.12 Écrire la fonction `fusion : 'a list -> 'a list -> 'a list` telle que si `l1` et `l2` sont deux listes triées par ordre croissant, l'appel `fusion l1 l2` renvoie une troisième liste triée contenant exactement les éléments de `l1` et `l2`.

```

1 || # fusion [10;15;20] [8;12;14;15;22];;
2 || - : int list = [8; 10; 12; 14; 15; 15; 20; 22]

```

Q.13 Écrire la fonction `separer : 'a list -> 'a list * 'a list` telle que `separer l` scinde la liste `l` en deux listes de tailles égales à 1 près, et dont la concaténation contient exactement les éléments de `l`.

```

1 || # separer [0;1;2;3;4];;
2 || - : int list * int list = ([4; 2; 0], [3; 1])

```

Q.14 Écrire la fonction `tri : 'a list -> 'a list` qui réalise le tri fusion d'une liste par ordre croissant.

```

1 | # tri [[4; 4]; [5; 6]; [5; 9]; [1; 4]; [11; 6]; [7; 2];
2 |       [4; 1]; [7; -1]; [8; 5]; [1; 8]; [13; 1]; [0; 0]];;
3 | - : int list list =
4 | [[0; 0]; [1; 4]; [1; 8]; [4; 1]; [4; 4]; [5; 6]; [5; 9];
5 | [7; -1]; [7; 2]; [8; 5]; [11; 6]; [13; 1]]

```

Q.15 On admet que les fonctions `fusion` et `separer` terminent et sont correctes et que leurs complexités sont linéaires en la taille de leur(s) argument(s).

Donner sous la forme $O(f(n))$ et en justifiant la complexité de l'appel `tri 1` pour une liste de taille n et f une fonction à déterminer.

2.2 Algorithme de balayage

Nous avons vu en DM l'algorithme du papier cadeau dont le coût est en $O(kn)$ où n est la taille du nuage et k le nombre de sommets de l'enveloppe convexe. Nous décrivons maintenant 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 *pires*, 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 *dépilage* consiste à retirer le sommet de la pile.

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 donnés par une liste de listes de deux coordonnées (voir 1).

On rappelle la définition suivante :

Définition 2. Étant donnés trois points p_i, p_j, p_k du nuage P , distincts ou non, l'*orientation* de p_i, p_j, p_k vaut 0 si les trois points sont alignés, +1 si l'angle $(\overrightarrow{p_0 p_1}, \overrightarrow{p_0 p_2})$ est orienté dans le sens direct et -1 s'il est orienté négativement.

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

Voir l'exemple figure 2.

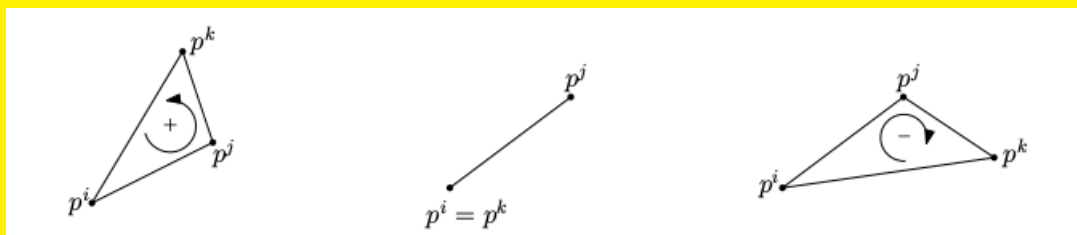


FIGURE 2 – Test d'orientation sur la séquence (p_i, p_j, p_k) : positif à gauche, nul au centre, négatif à droite.

Q.16 Calcul d'orientation :

- (a) Écrire la fonction `oriente : int list -> int list -> int list -> int` qui prend en paramètres 3 points p_0, p_1, p_2 dans cet ordre et renvoie le résultat $(-1, 0$ ou $+1)$ du test d'orientation sur la séquence (p_0, p_1, p_2) .

Rappel : il s'agit juste d'étudier le signe du déterminant $\det(\overrightarrow{p_0p_1}, \overrightarrow{p_0p_2})$.

- (b) Donner sa complexité.

À partir de maintenant, on supposera que les points fournis en entrée sont triés par abscisse croissante.

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 3.

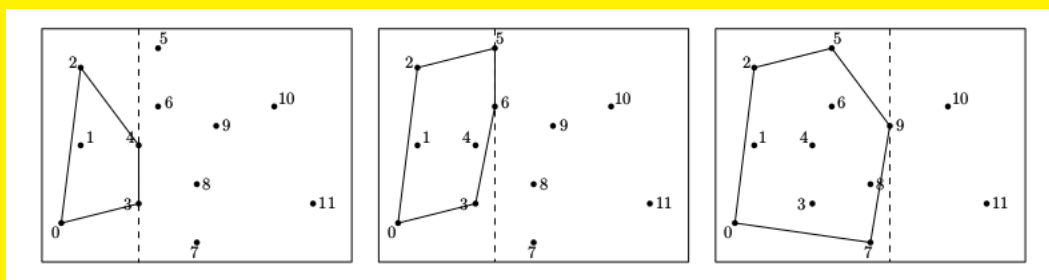


FIGURE 3 – Diverses étapes dans la procédure de balayage. La droite de balayage est en tirets.

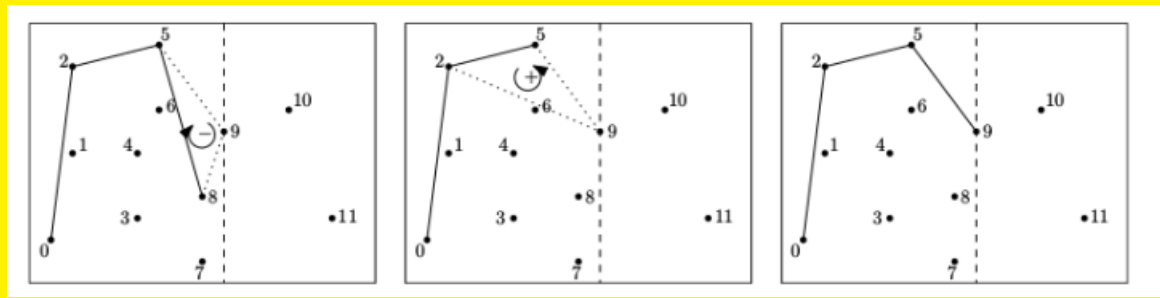
Plus précisément, l'algorithme visite chaque point de P une fois, par ordre croissant d'abscisse (donc par ordre croissant d'indice de colonne dans le tableau `tab` car celui-ci est trié). À chaque nouveau point p_i visité, il met à jour le bord de l'enveloppe convexe du sous-nuage $\{p_0, \dots, p_i\}$ 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 $\text{Conv}\{p_0, \dots, p_i\}$ 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 $\text{Conv}\{p_0, \dots, p_i\}$ située au-dessous (p_0 et p_i compris). Le bord de $\text{Conv}\{p_0, \dots, p_{n-1}\}$ est donc constitué de l'union de ces deux enveloppes, après suppression des doublons de p_i .

Informaticquement, 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).

Par exemple, dans le cas du nuage P de la figure 3 gauche, le sous-nuage $\{p_0, p_1, p_2, p_3, p_4\}$ a pour enveloppe supérieure `es` la pile `[p4; p2; p0]` et pour enveloppe inférieure `ei` la pile `[p4; p3; p0]`. 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 `[p0; p3; p4; p2; p0]`.

La mise à jour de l'enveloppe supérieure est illustrée dans la figure 4 : 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 (voir la définition 2 pour rappel de l'orientation), 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.

FIGURE 4 – Mise à jour de l'enveloppe supérieure lors de la visite du point p_9

Q.17 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.

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

Q.18 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.

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

Q.19 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).

```
1 # nuage;;
2 - : int list list =
3 [[0; 0]; [1; 4]; [1; 8]; [4; 1]; [4; 4]; [5; 9];
4 [5; 6]; [7; -1]; [7; 2]; [8; 5]; [11; 6]; [13; 1]]
5 # deuxEnv nuage;;
6 - : int list list * int list list =
7 ([[13; 1]; [7; -1]; [0; 0]],
8 [[13; 1]; [11; 6]; [5; 9]; [1; 8]; [0; 0]])
9 # deuxEnv [[1;2];[3;4]];;
10 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.20 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` ?

Démonstration. Exactement une fois

□

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

Démonstration. 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`.

Démonstration. Chaque point étant empilé une fois et dépilé au plus 1 fois, cela fait au plus 2 telles opérations 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.

Démonstration. Il y a moins de calculs d'orientations que de calculs d'empilement/dépilage.

Par exemple, si je dois retirer deux points pour empiler le 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. □

- (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 .

Démonstration. 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 $O(n)$. □

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

- Q.21** 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.

```
1 | # convGraham nuage;;
2 | - : int list list =
3 | [[0; 0]; [7; -1]; [13; 1]; [11; 6]; [5; 9]; [1; 8]; [0; 0]]
```

Appendice

- On peut utiliser avec profit dans ce devoir le *principe des tiroirs*. Il s'agit d'un énoncé très simple des mathématiques qui formalise l'évidence suivante : si vous avez 10 caleçons et que

vosre commode comporte 9 tiroirs, alors il y a au moins un tiroir qui contient 2 caleçons ! Mathématiquement ceci se traduit de la façon suivante : si E et F sont des ensembles avec F fini et $\text{card}(E) > \text{card}(F)$, et si $f : E \rightarrow F$ est une fonction, alors elle ne peut pas être injective (et donc il existe deux éléments qui ont la même image par f).

- Pour récupérer une exception sans bloquer le programme on utilise la construction

`try ... with ...`.

Par exemple, le code de `List.tl` contient une instruction `failwith "hd"`. On encapsule alors l'appel à `List.hd` :

```
1 | # try
2 |   List.hd [1;2]
3 | with Failure _ -> 0;;
4 | - : int = 1
5 | # try
6 |   List.hd [] (*Une exception est soulevée*)
7 | with Failure _ -> 0;; (*exception traité*)
8 | - : int = 0
```