Devoir ITC n°3

Enveloppe convexe dans le plan

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 se place dans le cas d'un nuage de points P ; on définit alors $\mathrm{Conv}(P)$ comme un polygône convexe dont les sommets appartiennent à P, tel que tous les points de P soient compris dans la surface définie par $\mathrm{Conv}(P)$, comme illustré dans la figure 1.

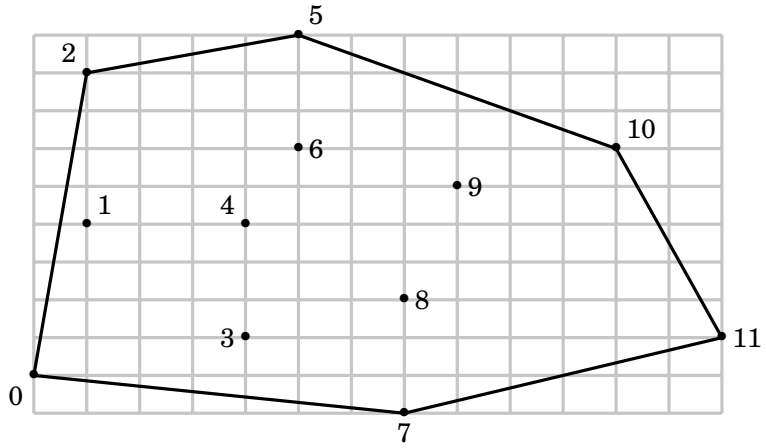

Figure 1 : Un nuage de points, numérotés de 0 à 11, et le bord de son enveloppe convexe

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 écrire deux algorithmes de calcul du bord de l'enveloppe convexe d'un nuage de points P dans le plan affine. Le premier, dit **algorithme du paquet cadeau**, consiste à envelopper le nuage de points P progressivement en faisant pivoter une droite tout autour. Le deuxième, dit **de balayage**, consiste à balayer le plan horizontalement avec une droite verticale, tout en maintenant au fur et à mesure l'enveloppe convexe de la partie du nuage située à gauche de cette droite verticale. Les deux algorithmes sont illustrés respectivement dans les figures 3 et 5.

On s'intéressera à la complexité de ces deux algorithmes, en fonction des paramètres n le nombre total de points de P et m le nombre de points de P appartenant au bord de $\mathrm{Conv}(P)$. Rappelons

Dans toute la suite on supposera que le nuage de points P est de taille $n \geqslant 3$ 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 un tableau `tab` à 2 dimensions (liste de listes), comme dans l'exemple ci-contre qui contient les coordonnées du nuage de points de la figure 1, et qu'on représente donc en python comme suit :

i \ j	0	1	2	3	4	5	6	7	8	9	10	11
0	0	1	1	4	4	5	5	7	7	8	11	13
1	0	4	8	1	4	9	6	-1	2	5	6	1

```
tab = [[0, 1, 1, 4, 4, 5, 7, 7, 8, 11, 13], [0, 4, 8, 1, 4, 9, 6, -1, 2, 5, 6, 1]]
```

Précisons que les coordonnées, supposées entières, sont données dans une base orthonormée du plan, orientée dans le sens direct. La première ligne du tableau contient les abscisses, tandis que la deuxième contient les ordonnées. Ainsi, la colonne d'indice j contient les deux coordonnées du point d'indice j. Ce dernier sera nommé p_j dans la suite. On obtient donc la coordonnée i (0 ou 1) du point j par `tab[i][j]`.

I. Fonctions préliminaires

1 – Écrire une fonction `plus_bas(tab: list) -> int` qui prend en paramètre le tableau `tab` de taille $2 \times n$ et qui renvoie l'indice j du point le plus bas (c'est-à-dire de plus petite ordonnée) parmi les points du nuage P. En cas d'égalité, votre fonction devra renvoyer l'indice du point de plus petite abscisse parmi les points les plus bas. Sur le tableau exemple précédent, le résultat de la fonction doit être l'indice 7.

```
def plus_bas(tab):
    ind = 0             # initialisation : 0 est l'indice du minimum
    n = len(tab[0])     # nombre de points
    for i in range(n):  # on teste chaque indice pour mettre à jour le minimum
        if tab[1][i] < tab[1][ind] or (tab[1][i] == tab[1][ind] and tab[0][i] < tab[0][ind]):
            ind = i
    return ind
```

Dans la suite nous aurons besoin d'effectuer un seul type de test géométrique : celui de l'orientation.

Définition : 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 (c'est-à-dire si deux au moins sont égaux, d'après l'hypothèse de position générale).

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 2. Cette aire est la moitié du déterminant de la matrice 2×2 formée par les coordonnées des vecteurs $\overrightarrow{p_i p_j}$ et $\overrightarrow{p_i p_k}$.

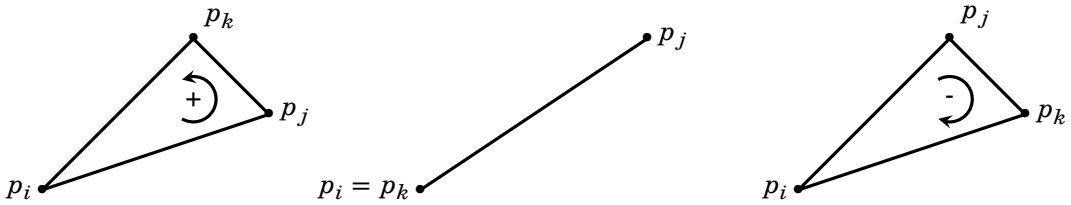

Figure 2 : Test d'orientation sur la séquence (p_i, p_j, p_k) : positif à gauche, nul au centre, négatif à droite.

2 – Sur le tableau exemple précédent, donner le résultat du test d'orientation pour les choix d'indices suivants :
- $i = 0, j = 3, k = 4$,
- $i = 8, j = 9, k = 10$

Si $i = 0, j = 3, k = 4$,

$$\frac{1}{2}\begin{vmatrix} 4 & 4 \\ 1 & 4 \end{vmatrix} = 6 > 0$$

Si $i = 8, j = 9, k = 10$,

$$\frac{1}{2}\begin{vmatrix} 1 & 4 \\ 3 & 4 \end{vmatrix} = -4 < 0$$

3 – Écrire une fonction `orient(tab: list, i: int, j: int, k: int) -> int` qui prend en paramètres le tableau `tab` et trois indices de colonnes, potentiellement égaux, et qui renvoie le résultat (-1, 0 ou $+1$) du test d'orientation sur la séquence (p_i, p_j, p_k) de points de P. Justifier que sa complexité est constante.

```python
def orient(tab, i, j, k):
    a, b = tab[0][j]-tab[0][i], tab[1][j]-tab[1][i]
    c, d = tab[0][k]-tab[0][i], tab[1][k]-tab[1][i]
    det = a*d - b*c  # calcul du déterminant
    if det == 0:     # cas pathologique
        return 0
    elif det > 0:
        return 1
    else:
        return -1
```

Les appels à un indice quelconque d'une liste Python sont en temps constant, donc l'ensemble de ces opérations l'est aussi.

II. Algorithme du paquet 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 $\text{Conv}(P)$. On commence par insérer le point de plus petite ordonnée (celui d'indice 7 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, $i = 10$ dans l'exemple de la figure 3. Pour un couple de points $(p_j, p_k) \in P\backslash\{p_i\}$, on considèrera que $p_j < p_k$ si `orient(tab,i,j,k)` $\leqslant 0$. Ainsi, le prochain point à insérer (le point d'indice 5 dans la figure 1) est l'élément maximum j ainsi défini :

$$\forall k \in P\backslash\{p_i\}, p_k < p_j$$

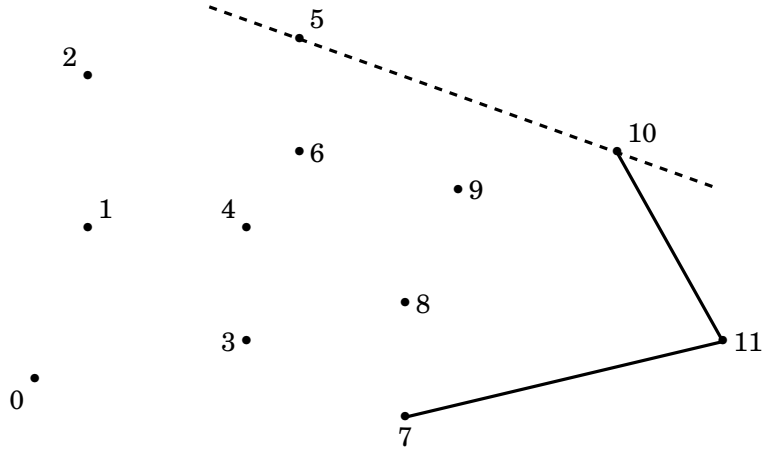

Figure 3 : Mise à jour du paquet cadeau après insertion du point p_{10}.

4 – Décrire à la main le déroulement d'une fonction `prochain_point` sur l'exemple de la figure 3. Plus précisément, indiquer la séquence des points de $P\backslash\{p_{10}\}$ considérés et la valeur de l'indice du maximum à chaque itération.

On note i le point considéré dans la séquence 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11 ; on initialise la boucle avec $sup = 0$. On note f la fonction *orient*

- $i = 0, sup = 0 : f(tab, 10, 0, 0) = 0 \Rightarrow sup$ inchangé
- $i = 1, sup = 0 : f(tab, 10, 1, 0) = 1 \Rightarrow sup \leftarrow 1$
- $i = 2, sup = 1 : f(tab, 10, 2, 1) = 1 \Rightarrow sup \leftarrow 2$
- $i = 3, sup = 2 : f(tab, 10, 3, 2) = -1 \Rightarrow sup$ inchangé
- $i = 4, sup = 2 : f(tab, 10, 4, 2) = -1 \Rightarrow sup$ inchangé
- $i = 5, sup = 2 : f(tab, 10, 5, 2) = 1 \Rightarrow sup \leftarrow 5$
- $i = 6, sup = 5 : f(tab, 10, 3, 5) = -1 \Rightarrow sup$ inchangé
- $i = 7, sup = 5 : f(tab, 10, 3, 5) = -1 \Rightarrow sup$ inchangé
- $i = 8, sup = 5 : f(tab, 10, 3, 5) = -1 \Rightarrow sup$ inchangé
- $i = 9, sup = 5 : f(tab, 10, 3, 5) = -1 \Rightarrow sup$ inchangé

- $i = 11, sup = 5 : f(tab, 10, 3, 5) = -1 \Rightarrow sup$ in- changé

---

5 – Écrire la fonction `prochain_point(tab: list, i: int) -> int`, qui prend en paramètre le tableau `tab` de taille $2 \times n$ ainsi que l'indice i du point inséré en dernier dans le paquet cadeau, et qui renvoie l'indice du prochain point à insérer. Le temps d'exécution de votre fonction doit être en $\mathcal{O}(n)$ pour tous n et i ; le justifier.

```python
def prochain_point(tab,i):
    n = len(tab[0])
    sup = 0
    if i == 0:  # cas pathologique : si i == sup, le test renvoie toujours 0 ! Il faut gérer si i = 0
        sup = 1
    for k in [l for l in range(n) if l != i]:  # on exclut i à la construction
        if orient(tab, i, k, sup) > 0:
            sup = k
    return sup
```

La fonction effectue $n-1$ boucles for, et ce indépendemment de la valeur de i ; puisque chaque boucle contient au plus une comparaison et une affectation, on peut en déduire que le temps d'exécution de cette fonction est majoré par une constante fois n.

On peut maintenant combiner la fonction `prochain_point` avec la fonction `plus_bas` de la question 1 pour calculer le bord de l'enveloppe convexe de P. On commence par insérer le point p_i d'ordonnée la plus basse, puis on itère le processus de mise à jour du paquet cadeau jusqu'à ce que le prochain point à insérer soit de nouveau p_i. À ce moment-là on renvoie le paquet cadeau comme résultat sous forme de liste, sans insérer p_i une seconde fois.

6 – Écrire une fonction `conv_Jarvis(tab: int) -> list` qui prend en paramètre le tableau `tab` de taille $2 \times n$ représentant le nuage P, et qui renvoie une liste contenant les indices des sommets du bord de l'enveloppe convexe de P, sans doublon. Le temps d'exécution de votre fonction doit être $\mathcal{O}(nm)$, où m est le nombre de points de P situés sur le bord de $\text{Conv}(P)$. On justifiera cette complexité.

```python
def conv_Jarvis(tab):
    enveloppe = [plus_bas(tab)]  # initialisation avec plus_bas
    suivant = prochain_point(tab, enveloppe[-1])
    while suivant != enveloppe[0]:  # on continue tant que l'on a pas bouclé
        enveloppe.append(suivant)
        suivant = prochain_point(tab,enveloppe[-1])
    return enveloppe
```

La boucle while fait exactement m boucles, et `prochain_point` a un temps de calcul majoré par une constante fois n, donc la fonction `conv_Jarvis` a un temps de calcul majoré par mn, a une constante près.

III. Intermède : pile d'entiers et tri

Dans la suite nous aurons besoin d'utiliser des piles d'entiers, dont on rappelle la définition ci-dessous :

Définition : Une pile d'entiers est une structure de données LIFO (pour Last In, First Out) permettant de stocker des entiers et d'effectuer les opérations suivantes en temps constant (indépendant de la taille de la pile) :

- créer une nouvelle pile vide,
- déterminer si la pile est vide,
- insérer un entier au sommet de la pile,
- retirer l'entier au sommet de la pile.

Nous supposerons fournies les fonctions suivantes du module `queue`, qui réalisent les opérations ci-dessus et s'exécutent chacune en temps constant :

- `LifoQueue()`, qui ne prend pas d'argument et renvoie une pile vide,
- `p.empty()`, qui prend une pile `p` en argument et renvoie `True` ou `False` suivant que `p` est vide ou non,
- `p.put(i)`, qui prend un entier `i` et une pile `p` en argument, insère `i` au sommet de `p` et ne renvoie rien,
- `p.get()`, qui prend une pile `p` (supposée non vide) en argument, supprime l'entier au sommet de `p` et renvoie sa valeur.

Dans la suite il est demandé aux candidats de manipuler les piles uniquement au travers de ces fonctions, sans aucune hypothèse sur la représentation effective des piles en mémoire.

7 – Écrire une fonction `deux_éléments(p: LifoQueue) -> bool` qui prend une pile `p` en paramètre et renvoie `True` si la pile contient **au moins** deux éléments, et `False` si elle n'en contient que 0 ou 1. La pile doit être inchangée après exécution de la fonction.

```
def deux_éléments(p):
    if p.empty():
        return False
    else:
        x = p.pop()
        if p.empty():
            rép = False
        else:
            rép = True
        p.put(x)
        return rép
```

8 – Écrire une fonction `paire_sommet(p: LifoQueue) -> tuple` qui prend une pile `p` contenant au moins deux éléments en paramètre et renvoie la paire « (dernier élément, avant-dernier élément) » sans modifier la pile.

```
def paire_sommet(p):
    dernier = p.get()
    avant_dernier = p.get()
    p.put(avant_dernier)
    p.put(dernier)
    return (dernier, avant_dernier)
```

9 – Écrire une procédure `transvase{d: LifoQueue, a: LifoQueue} -> None` qui prend deux piles `d` et `a` et qui vide la pile `d` dans la pile `a` en l'inversant, comme sur la figure 4.

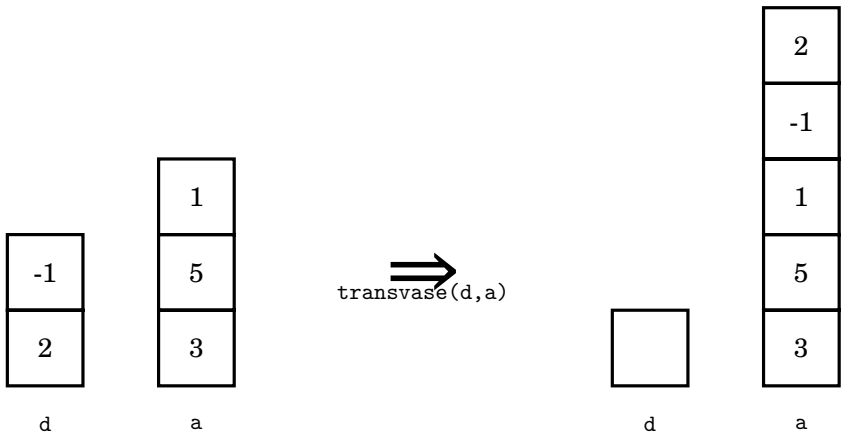

Figure 4 : Exemple d'effet de la fonction `transvase(d,a)`

```
def transvase(d, a):
    while not d.empty():
        a.put(d.get())
```

Nous aurons également besoin dans la suite de trier le tableau des points par abcisse croissante.

10 – Citer un algorithme faisant un tri d'un tableau en complexité $\mathcal{O}(n \log n)$ (complexité garantie ou moyenne). Écrire une fonction `tri(tab)` permettant de trier ainsi un tableau de nombres `tab`. On pourra faire l'hypothèse que la syntaxe `tab[i:j]` donne accès au-sous tableau lui-même et n'en fait pas une copie, comme si la liste était un tableau NumPy.

On propose le tri rapide :

```
def partitionner(tab):
    n = len(tab)
    piv = tab[n-1]
    i, j = 0, n-2  # deux pointeurs : gauche et droite
    while j >= i:
        if tab[i] <= piv:
            i += 1
        elif tab[j] >= piv:
            j -= 1
        else:
            tab[i], tab[j] = tab[j], tab[i]
    # on place le pivot au bon endroit
    tab[i], tab[n-1] = tab[n-1], tab[i]
    return i

def tri_rapide(tab):
    n = len(tab)
    if n > 1:  # cas de base (inversé)
        j = partitionner(tab)
        tri_rapide(tab[0:j])
        tri_rapide(tab[j+1:n])
```

ou le tri fusion

```
def fusionner(t1, t2):
    n1, n2 = len(t1), len(t2)
    i1, i2 = 0, 0    # compteurs pour savoir quel est l'élément courant
    tr = [0] * (n1+n2)  # tableau à renvoyer
    t1.append(float('inf'))  # sentinelle
    t2.append(float('inf'))  # sentinelle
    for j in range(n1+n2):
        if t1[i1] <= t2[i2]:
            tr[j], i1 = t1[i1], i1+1
        else:
            tr[j], i2 = t2[i2], i2+1
    return tr

def tri_fusion(tab):
    n = len(tab)
    if n == 1:  # condition d'arrêt
        return tab
    t1 = tri_fusion(tab[:n//2])  # diviser + régner sur une moitié
    t2 = tri_fusion(tab[n//2:])  # diviser + régner sur l'autre moitié
    return fusionner(t1, t2)     # combiner
```

11 – Que faudrait-il modifier dans la fonction précédente pour trier le tableau de points (liste de deux listes de taille n, comme celui donné en début d'énoncé) par abscisse croissante ?

Il faudrait changer les tests du type `t1[i1] <= t2[i2]` en `t1[0][i1] <= t2[0][i2]` pour bien comparer les abcisses, et les affectations du type `tr[j] = t1[i1]` par `tr[j] = [t1[0][i1], t1[1][i1]]` .

À partir de maintenant, on supposera que les points fournis en entrée sont triés par abscisse croissante, comme c'est le cas dans l'exemple du tableau `tab` donné au début du sujet.

IV. Balayage

Cet algorithme a été proposé par R. Graham en 1972. Nous allons écrire la variante (plus simple) proposée par A. Andrew quelques années plus tard.

La première étape consiste à trier les n points du nuage P par ordre croissant d'abscisse, en conservant tous les points de même abscisse dans un ordre arbitraire. Cela est supposé fait avec la fonction de tri proposée précédemment.

L'idée de l'algorithme est alors 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 5.

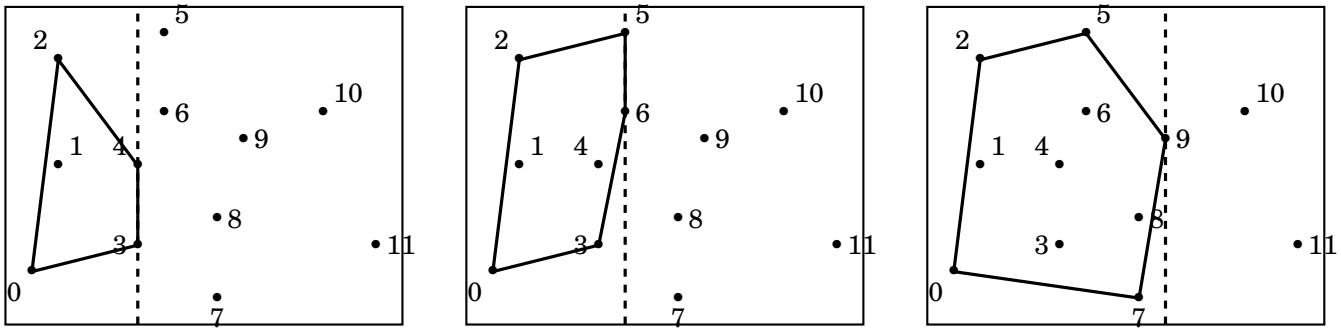

Figure 5 : Diverses étapes dans la procédure de balayage. La droite de balayage est en tirets.

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). A 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 $\text{Conv}\{p_0, \dots, p_i\}$ est donc constitué de l'union de ces deux enveloppes, après suppression des doublons de p_0 et p_i.

Par exemple, dans le cas du nuage P de la figure 5 gauche, le sous-nuage $\{p_0, p_1, p_2, p_3, p_4\}$ a pour enveloppe supérieure la séquence (p_0, p_2, p_4) et pour enveloppe inférieure la séquence (p_0, p_3, p_4), le bord de son enveloppe convexe étant donné par la séquence (p_0, p_3, p_4, p_2).

Informatiquement, les indices des sommets des enveloppes inférieure et supérieure seront stockés dans deux piles d'entiers séparées, `ei` (pour enveloppe inférieure) et `es` (pour enveloppe supérieure).

La mise à jour de l'enveloppe supérieure est illustrée dans la figure 6 : 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) forment une séquence (p_9, p_8, p_5) d'orientation négative (voir la définition pour rappel de l'orientation), on dépile l'indice situé au sommet de `es` (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 indice dans la pile. L'indice du 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.

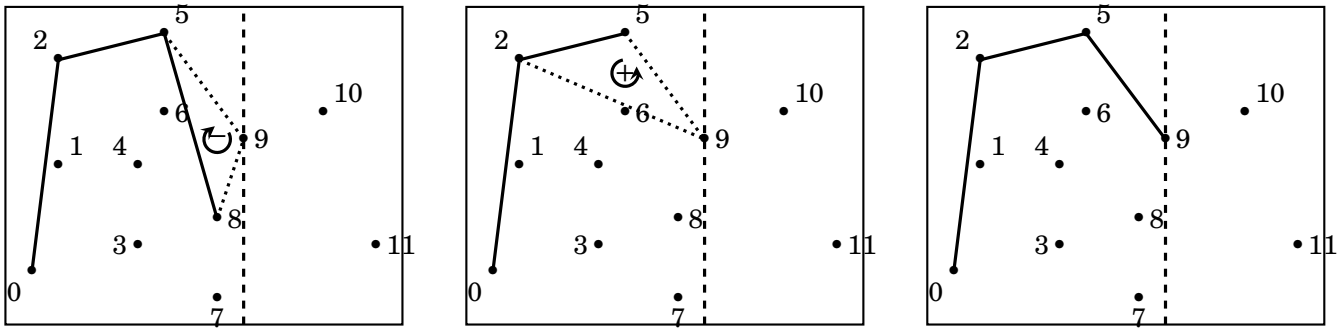

Figure 6 : Mise à jour de l'enveloppe supérieure lors de la visite du point p_9.

12 – Écrire une fonction `majES(tab: list, es: LifoQueue, i:int)` qui prend en paramètre le tableau `tab` ainsi que la pile `es` et l'indice i du point à visiter, et qui met à jour l'enveloppe supérieure du sous-nuage. La complexité doit être $\mathcal{O}(i)$; justifier cette complexité.

```
def majES(tab,es,i):
    if not deux_éléments(es):  # s'il n'y a qu'un élément dans la pile, on empile simplement
        es.put(i)
    else:
        dernier, avtdernier = paire_sommet(es)
```

```
        # on teste l'orientation ET qu'il reste assez de points
        while orient(tab, i, dernier, avtdernier) < 0 and deux_éléments(es):
            es.get()  # on dépile
            if deux_éléments(es):  # cas où on a dépilé l'avant-dernier élément
                dernier, avtdernier = paire_sommet(es)
        es.put(i)  # on empile le nouvel élément
```

Dans le pire cas, la boucle while vide la pile ; puisque la pile contient au plus i éléments (on tente de remplir la pile en visitant les points dans l'ordre, donc on les empile dans l'ordre), cette boucle while fera au plus i itérations. Puisque chaque tour de boucle contient uniquement des opérations en temps constant (dépiler, tester la présence de deux éléments, observer les deux éléments du haut), le temps d'éxécution de cette fonction est majoré par une constante fois i.

On suppose ensuite écrite une fonction `majEI(tab, ei, i)` qui effectue la mise à jour de l'enveloppe inférieure, avec le même temps d'exécution.

13 – Écrire maintenant une fonction `conv_Graham(tab: list) -> int` qui prend en paramètre le tableau `tab` de taille $2 \times n$ représentant le nuage P, et qui effectue le balayage des points de P comme décrit précédemment. On supposera les colonnes du tableau `tab` déjà triées par ordre croissant d'abscisse. La fonction doit renvoyer une pile `s` contenant les indices des sommets du bord de $\text{Conv}(P)$ triés dans l'ordre positif d'orientation, à commencer par le point p_0. Par exemple, sur le nuage de la figure 1, le résultat de la fonction `conv_Graham` doit être la pile `s` contenant la suite d'incides 0, 7, 11, 10, 5, 2 dans cet ordre, l'indice 0 se trouvant au fond de la pile `s` et l'indice 2 au sommet de s.

```
def conv_Graham(tab):
    # initialisation des enveloppes
    n, env_i, env_s = len(tab[0]), LifoQueue(), LifoQueue()
    env_i.put(0)
    env_s.put(0)
    for i in range(1, n):  # on visite chaque point
        majEI(tab, env_i, i)
        majES(tab, env_s, i)
    env_s.get()              # on enlève le dernier point de l'enveloppe supérieure
    transvase(env_s, env_i)  # on transvase
    env_i.get()              # le point de départ est en trop au sommet de la pile
    return env_i
```

---

14 – Quel est la complexité de l'algorithme de balayage décrit précédemment ? En déduire la complexité totale de l'algorithme de Graham-Andrew.

La boucle for effectue $n-1$ boucles. Il faut alors remarquer que si un élément est empilé dans l'une des enveloppes, il ne pourra être dépiler au plus qu'une fois ; chaque élément du nuage de points conduira donc au plus à deux opérations dans la boucle for de la fonction `conv_Graham`. Les opérations sur l'ensemble de la boucle for sont donc majorées par une constante fois n. Finalement, la partie la plus coûteuse de l'algorithme est l'étape de tri, qui selon l'énoncé peut être fait en temps $n \log n$. Au final, le temps d'exécution total de l'algorithme de Graham-Andrew est $\mathcal{O}(n \log n)$
