Important : les fichiers mentionnés dans ce sujet sont disponibles sur la page web du cours.
Ce sujet est inspiré de l'épreuve X/ENS info B 2015, adaptée pour le langage C par Marc de Falco.

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 dit 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 $Conv(P)$, est le plus petit convexe contenant P. Dans le cas où P est un ensemble fini (appelé nuage de points), le bord de $Conv(P)$ est un polygone convexe dont les sommets appartiennent à P, comme illustré dans la figure 1.

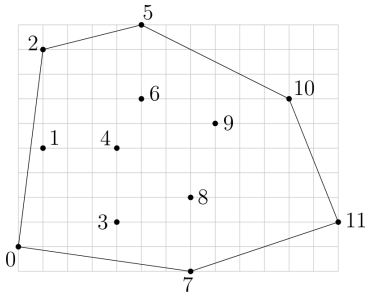

FIGURE 1 – Un nuage de points, numérotés de 0 à 11, et le bord de son enveloppe convexe.

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 4.

Le temps d'exécution du premier algorithme est majoré par une constante fois nm, celui du deuxième par une constante fois $n \log n$, où n désigne le nombre total de points de P et m le nombre de points de P appartenant au bord de $Conv(P)$. 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, le candidat n'aura pas à justifier des temps de calcul de ses programmes. Toutefois, il devra veiller à ce que ces derniers ne dépassent pas les bornes prescrites.

On va représenter les points par des éléments du type `point` :

```c
struct point {
    int x;
    int y;
};
typedef struct point point;
```

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 un tableau `tab` de `point`, comme dans l'exemple ci-dessous qui contient les coordonnées du nuage de points de la figure 1 :

```c
point tab[] = {
    { 0, 0 }, { 1, 4 }, { 1, 8 }, { 4, 1 }, { 4, 4 },
    { 5, 9 }, { 7, -1 }, { 7, 2 }, { 8, 5 }, { 11, 6 }, { 13, 1 }
}; // initialisation directe des struct comme pour les tableaux
```

Précisons que les coordonnées, supposées entières, sont données dans une base orthonormée du plan, orientée dans le sens direct.

A - Préliminaires

Question 1. Écrire une fonction `int plus_bas(point *tab, int n);` qui prend en paramètre le tableau `tab` de n points 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.

Question 2. Prouver la correction de cette fonction.

Sur le tableau exemple précédent, le résultat de la fonction doit être l'indice 7.

Dans la suite nous aurons besoin 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 (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 vaut :

$$\frac{1}{2}\left((y_k - y_i)(x_j - x_i) - (x_k - x_i)(y_j - y_i)\right)$$

où p_a a pour coordonnées (x_a, y_a). **Comme on travaille avec des entiers, on évitera de faire la division par 2 qui peut aboutir à 0 et ne préserve pas le signe strict.**

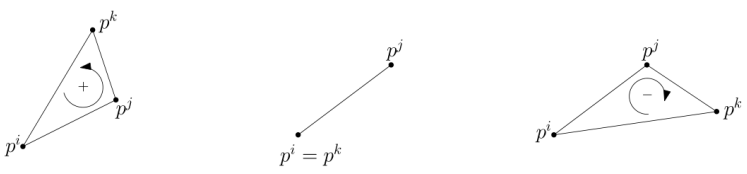

FIGURE 2 – Test d'orientation sur la séquence (p_i, p_j, p_k) : positif à gauche, nul au centre, négatif à droite.

Question 3. Sur le tableau exemple précédent, donner le résultat du test d'orientation pour les choix d'indices suivants :

1. i = 0, j = 3, k = 4,
2. i = 8, j = 9, k = 10.

Question 4. Écrire une fonction `int orient(point *tab, int i, int j, int k);` qui prend en paramètres le tableau *tab* et trois indices, 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.

B - 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 $Conv(P)$. 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.

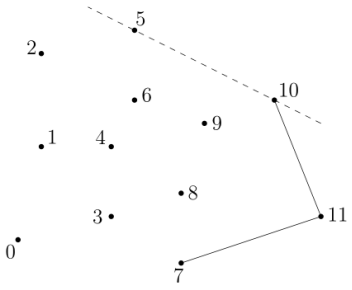

FIGURE 3 – Mise à jour du paquet cadeau après insertion du point p_0.

Considérons la relation binaire $\preceq$ définie sur l'ensemble $P\backslash\{p_i\}$ par :

$$p_j \preceq p_k \iff \texttt{orient(tab, i, j, k)} \leq 0$$

Question 5. Justifier brièvement le fait que $\preceq$ est une relation d'ordre total sur l'ensemble $P\backslash\{p_i\}$.

Ainsi, le prochain point à insérer (le point d'indice 5 dans la figure 3) est l'élément maximum pour la relation d'ordre $\preceq$. Il peut se calculer en temps linéaire (c'est-à-dire majoré par une constante fois n) par une simple itération sur les points de $P\backslash\{p_i\}$.

Question 6. Écrire une fonction `int prochain_point(point *tab, int n, int i);` qui prend en paramètre le tableau `tab` de 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 selon cette procédure. Le temps d'exécution de votre fonction doit être majoré par une constante fois n, pour tous n et i. La constante doit être indépendante de n et i, et on ne demande pas de la préciser.

Question 7. Décrire à la main le déroulement de la 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 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 sans insérer p_i une seconde fois.

Un détail technique : comme la taille du paquet cadeau augmente peu à peu lors du processus, et qu'à la fin elle peut être petite par rapport au nombre n de points de P, nous stockerons les indices des points du paquet cadeau dans un tableau de n entiers en renvoyant le nombre de points de l'enveloppe.

Par exemple, sur le nuage de la figure 1, on placera $7, 11, 10, 5, 2, 0$ dans le tableau d'enveloppe, sans toucher aux indices 7 à 11 de ce tableau et on renverra la valeur 6 correspondant au nombre de points.

Question 8. Écrire une fonction `int conv_jarvis(point *tab, int n, int *env);` qui prend en paramètre le tableau `tab` de taille n représentant le nuage P, un tableau de même taille `env` et qui place dans `env` les indices du bord de l'enveloppe convexe de P, sans doublon et renvoie le nombre de points qui le constitue. Le temps d'exécution de votre fonction doit être majoré par une constante fois nm, où m est le nombre de points de P situés sur le bord de $Conv(P)$ (c'est la valeur renvoyée par la fonction).

Question 9. Justifier brièvement le temps d'exécution de l'algorithme du paquet cadeau.

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

Définition 2 *Une pile d'entiers est une structure de données 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,*
- *déterminer la valeur de l'entier au sommet de la pile,*
- *retirer l'entier au sommet de la pile.*

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

- `stack new_stack();` qui ne prend pas d'argument et renvoie une pile vide ;
- `bool is_empty(stack s);` qui prend une pile `s` en argument et renvoie la valeur `true` ou `false` suivant que `s` est vide ou non ;
- `void push(int i, stack s);` qui prend un entier `i` et une pile `s` en argument, insère `i` au sommet de `s` (c'est-à-dire à la fin de la liste), et ne renvoie rien ;
- `int top(stack s);` qui prend une pile `s` (supposée non vide) en argument et renvoie la valeur de l'entier au sommet de `s` (c'est-à-dire le dernier élément empilé).
- `int pop(stack s);` qui prend une pile `s` (supposée non vide) en argument, supprime l'entier au sommet de `s` et renvoie sa valeur.
- `void destroy_stack(stack s);` qui libère l'espace utilisé par la pile `s`.

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.

Vous pourrez trouver une implémentation de pile avec cette interface dans `stack.h` et `stack.c` de l'archive principale du DM.

Si on écrit

```
stack s = new_stack();
for(int i = 0; i < 3; i++)
{
    push(i, s);
    printf("%d\n", top(s));
}

for(int i = 0; i < 3; i++)
    printf("%d\n", pop(s));
destroy_stack(s);
```

On va alors afficher 0 1 2 à mesure qu'on empile, puis 2 1 0 en dépilant.

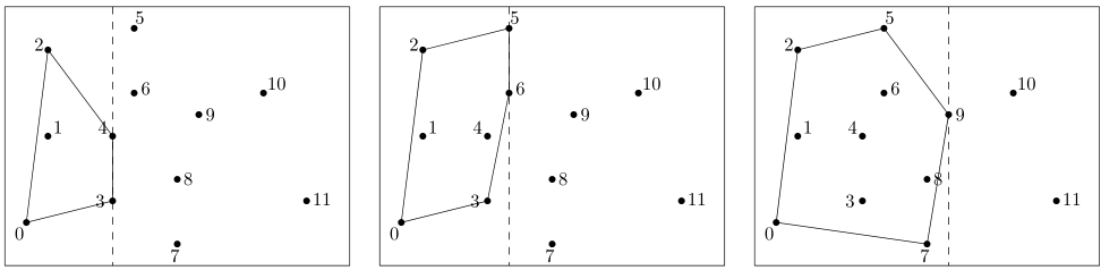

FIGURE 4 – Diverses étapes dans la procédure de balayage. La droite de balayage est en tirets.

D - Algorithme de 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.

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.

L'idée de l'algorithme 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 4.

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 $Conv\{p_0, \dots, p_i\}$ 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 $Conv\{p_0, \dots, p_i\}$ située au-dessous (p_0 et p_i compris). Le bord de $Conv\{p_0, \dots, p_i\}$ 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 4 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 5 : 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 1 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.

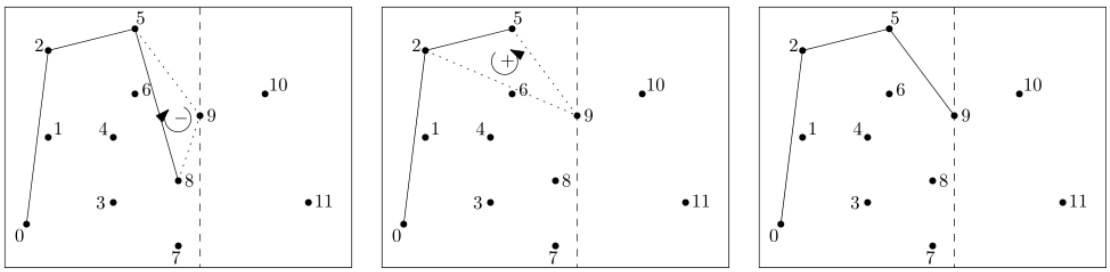

FIGURE 5 – Mise à jour de l'enveloppe supérieure lors de la visite du point p_9

Question 10. Écrire une fonction `void maj_es(point *tab, stack es, int i);` qui prend en paramètre le tableau `tab` ainsi que la pile `es`, non vide, et l'indice `i` du point à visiter, et qui met à jour l'enveloppe supérieure du sous-nuage. Le temps d'exécution de votre fonction doit être majoré par une constante fois `i`.

Question 11. Écrire une fonction `void maj_ei(point *tab, stack ei, int i);` qui effectue la mise à jour de l'enveloppe inférieure, avec le même temps d'exécution.

Question 12. Écrire maintenant une fonction `stack conv_graham(point *tab, int n);` qui prend en paramètre le tableau `tab` de taille n représentant le nuage P, et qui effectue le balayage des points de P comme décrit précédemment. On supposera le tableau `tab` déjà trié par ordre croissant d'abscisse. La pile renvoyée devra contenir les indices des sommets du bord de $Conv(P)$ 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 `env` contenant la suite d'indices 0, 7, 11, 10, 5, 2 dans cet ordre, l'indice 0 se trouvant au fond de la pile `env` et l'indice 2 au sommet de `env`.

Question 13. Analyser brièvement le temps d'exécution de l'algorithme de balayage décrit précédemment, en supposant une fois encore que les points du nuage fourni en entrée sont déjà triés par abscisse croissante. En déduire que le temps d'exécution total de l'algorithme de Graham-Andrew est bien majoré par une constante fois $n \log n$.