Cette épreuve est constituée de deux problèmes indépendants.

Problème n ° 1

Notations.
Pour m et n deux entiers naturels, $[\![m;n]\!]$ désigne l'ensemble des entiers k tels que $m \leqslant k \leqslant n$.

La résolution d'une grille de Sudoku est une gymnastique du cerveau qui peut être assimilée à un décodage correcteur d'effacement. En effet, à partir d'une grille presque vide, il est possible (pour une grille bien faite) de la compléter d'une unique manière.
L'objectif de cet exercice est de mettre en œuvre deux méthodes permettant de compléter une grille de Sudoku, l'une naïve, et l'autre par backtracking.

Une grille de Sudoku est une grille de taille 9 × 9, découpée en 9 carrés de taille 3×3. Le but est de la remplir avec des chiffres de $[\![1;9]\!]$, de sorte que chaque ligne, chaque colonne et chacun des 9 carrés de taille 3 × 3 contienne une et une seule fois chaque entier de $[\![1;9]\!]$. On dira alors que la grille est complète. En pratique, certaines cases sont déjà remplies et on fera l'hypothèse que le Sudoku qui nous intéresse est bien écrit, c'est-à-dire qu'il possède une unique solution.
On représente en Python une grille de Sudoku par une liste de taille 9 × 9, c'est-à-dire une liste de 9 listes de taille 9, dans laquelle les cases non remplies sont associées au chiffre 0. Ainsi, la grille suivante est représentée par la liste ci-contre :

	6					2		5
4			9	2	1			
	7				8			1
					5			9
6	4						7	3
1			4					
3			7				6	
			1	4	6			2
2		6					1	

```
L= [[0,6,0,0,0,0,2,0,5],[4,0,0,9,2,1,0,0,0],
    [0,7,0,0,0,8,0,0,1],[0,0,0,0,0,5,0,0,9],
    [6,4,0,0,0,0,0,7,3],[1,0,0,4,0,0,0,0,0],
    [3,0,0,7,0,0,0,6,0],[0,0,0,1,4,6,0,0,2],
    [2,0,6,0,0,0,0,1,0]]
```

Les 9 carrés de taille 3×3 sont numérotés du haut à gauche jusqu'en bas à droite. Ainsi, sur cette grille, le carré 0, en haut et à gauche, contient les chiffres 6, 4 et 7 ; le carré 1, en haut et au milieu, contient les chiffres 9, 2, 1 et 8 ; le carré 8, en bas et à droite, contient les chiffres 6, 2 et 1.

On rappelle que les lignes du Sudoku sont alors les éléments de `L` accessibles par `L[0],...,L[8]`. L'élément de la case (i, j) est accessible par `L[i][j]`.

Remarque : on fera bien attention, dans l'ensemble de ce sujet, aux indices des tableaux. Les lignes, ainsi que les colonnes, sont indicées de 0 à 8.

Partie A : Généralités

Résultats préliminaires

1. Montrer que si une grille de Sudoku est complète, alors pour chacune des lignes, chacune des colonnes et chacun des carrés de taille 3×3, la somme des chiffres fait 45. La réciproque est-elle vraie ?

2. Écrire une fonction `ligne_complete(L,i)` qui prend une liste Sudoku `L` et un entier i entre 0 et 8, et renvoie `True` si la ligne i du Sudoku `L` vérifie les conditions de remplissage d'un Sudoku, et `False` sinon.
On définit de même (on ne demande pas de les écrire) les fonctions `colonne_complete(L,i)` pour la colonne i et `carre_complet(L,i)` pour le carré i.

3. Écrire une fonction `complet(L)` qui prend une liste Sudoku `L` comme argument, et qui renvoie `True` si la grille est complète, `False` sinon.

Fonctions annexes

4. Compléter la fonction suivante `ligne(L,i)`, qui renvoie la liste des nombres compris entre 1 et 9 qui apparaissent sur la ligne d'indice i.

```
def ligne(L,i):
    chiffre = []
    for j in ....:
        if(....):
            chiffre.append(L[i][j])
    return chiffre
```

Ainsi, avec la grille donnée dans l'énoncé, on doit obtenir :

```
>>> ligne(L,0)
[6, 2, 5]
```

On définit alors, de la même manière, la fonction `colonne(L,j)` qui renvoie la liste des nombres compris entre 1 et 9 qui apparaissent dans la colonne j (on ne demande pas d'écrire son code).

5. On se donne une case (i, j), avec $(i, j) \in [\![0; 8]\!]^2$. Montrer que la case en haut à gauche du carré 3×3 auquel appartient la case (i, j) a pour coordonnées $\left(3 \times \left[\frac{i}{3}\right], 3 \times \left[\frac{j}{3}\right]\right)$, où $[x]$ représente la partie entière de x.

6. Compléter alors la fonction `carre(L,i,j)`, qui renvoie la liste des nombres compris entre 1 et 9 qui apparaissent dans le carré 3×3 auquel appartient la case (i, j).

```
def carre(L,i,j):
    icoin = 3*(i//3)
    jcoin = 3*(j//3)
    chiffre = []
    for i in range(....):
        for j in range(....):
            if(....):
                chiffre.append(L[i][j])
    return chiffre
```

On rappelle que si `x` et `y` sont des entiers, `x//y` renvoie le quotient de la division euclidienne de `x` par `y`. Ainsi, avec la grille donnée dans l'énoncé, on doit obtenir :

```
>>> carre(L,4,6)
[9, 7, 3]
>>>carre(L,4,5)
[5, 4]
```

7. Déduire des questions précédentes une fonction `conflit(L,i,j)` renvoyant la liste des chiffres que l'on ne peut pas écrire en case (i, j) sans contredire les règles du jeu. La liste renvoyée peut très bien comporter des redondances. On ne prendra pas en compte la valeur de `L[i][j]`.

8. Compléter enfin la fonction `chiffres_ok(L,i,j)` qui renvoie la liste des chiffres que l'on peut écrire en case (i, j).

```
def chiffres_ok(L,i,j):
    ok = []
    conflit = conflit(L,i,j)
    for k in ....:
        if ....:
            ok.append(k)
    return ok
```

Par exemple, avec la grille initiale :

```
>>> chiffres_ok(L,4,2)
[2, 5, 8, 9]
```

On pourra, dans la suite du sujet, utiliser les fonctions annexes définies précédemment.

Partie B : Algorithme naïf

Naïvement, on commence par compléter les cases n'ayant qu'une seule possibilité.
Nous prendrons dans la suite comme Sudoku :

```
M= [[2, 0, 0, 0, 9, 0, 3, 0, 0], [0, 1, 9, 0, 8, 0, 0, 7, 4],
    [0, 0, 8, 4, 0, 0, 6, 2, 0], [5, 9, 0, 6, 2, 1, 0, 0, 0],
    [0, 2, 7, 0, 0, 0, 1, 6, 0], [0, 0, 0, 5, 7, 4, 0, 9, 3],
    [0, 8, 5, 0, 0, 9, 7, 0, 0], [9, 3, 0, 0, 5, 0, 8, 4, 0],
    [0, 0, 2, 0, 6, 0, 0, 0, 1]]
```

9. A partir des fonctions annexes, écrire une fonction `nb_possible(L,i,j)`, indiquant le nombre de chiffres possibles à la case (i, j).

10. On souhaite disposer de la fonction `un_tour(L)` qui parcourt l'ensemble des cases du Sudoku et qui complète les cases dans le cas où il n'y a qu'un chiffre possible, et renvoie `True` s'il y a eu un changement, et `False` sinon. La liste `L` est alors modifiée par effet de bords.

Par exemple, en partant de la grille initiale `M` :

```
>>> un_tour(M)
True
>>> M
[[2, 0, 0, 0, 9, 0, 3, 0, 0], [0, 1, 9, 0, 8, 0, 5, 7, 4],
 [0, 0, 8, 4, 0, 0, 6, 2, 9], [5, 9, 0, 6, 2, 1, 4, 8, 7],
 [0, 2, 7, 0, 3, 8, 1, 6, 5], [0, 6, 1, 5, 7, 4, 2, 9, 3],
 [0, 8, 5, 0, 0, 9, 7, 3, 0], [9, 3, 6, 0, 5, 0, 8, 4, 2],
 [0, 0, 2, 0, 6, 0, 9, 5, 1]]
```

On propose la fonction suivante :

```
def un_tour(L):
    changement = False
    for i in range(1,9):
        for j in range (1,9):
            if (L[i][j] = 0):
                if (nb_possible(L,i,j) = 1):
                    L[i][j] = chiffres_ok(L,i,j)[1]
    return changement
```

Recopier ce code en en corrigeant les erreurs.

11. Écrire une fonction `complete(L)` qui exécute la fonction `un_tour` tant qu'elle modifie la liste, et renvoie `True` si la grille est complétée, et `False` sinon.

Partie C : Backtracking

La deuxième idée est de résoudre la grille par « Backtracking » ou « retour-arrière ». L'objectif est d'essayer de compléter la grille de Sudoku en testant les combinaisons, en commençant par la première case, et jusqu'à la dernière. Si on obtient un conflit avec les règles, on est obligé de revenir en arrière.
On va compléter la grille en utilisant l'ordre lexicographique, c'est-à-dire les cases $(0,0), \cdots, (0,8)$ puis $(1,0), (1,1), \cdots, (1,8), (2,0), \cdots$.

12. Écrire une fonction `case_suivante(pos)` qui prend une liste `pos` du couple des coordonnées de la case, et renvoie la liste du couple d'indices de la case suivante en utilisant l'ordre lexicographique, et qui renvoie `[9,0]` si `pos=[8,8]`. Par exemple :

```
>>> case_suivante([1,3])
[1, 4]
>>> case_suivante([8,8])
[9, 0]
```

La fonction principale va avoir la structure suivante :

```
def solution_sudoku(L):
    return backtracking(L,[0,0])
```

où `backtracking(L,pos)` est une fonction récursive qui doit renvoyer `True` s'il est possible de compléter la grille à partir des hypothèses faites sur les cases qui précèdent la case `pos`, et `False` dans le cas contraire. Ainsi :

- Si `pos` est la liste `[9, 0]`, la grille est complétée, et on renvoie `True` (cas d'arrêt).
- Si la case est déjà remplie (donnée initiale du Sudoku), on passe à la case suivante via un appel récursif.
- Sinon, on affecte un des chiffres possibles à la case, et on passe à la case suivante par un appel récursif.

13. Compléter le squelette de la fonction `backtracking(L,pos)` selon les règles précédentes.

```
def backtracking(L, pos):
    """
    pos est une liste désignant une case du sudoku,
    [0,0] pour le coin en haut à gauche.
    """
    if (pos==[9, 0]):
        .....
    i,j = pos[0],pos[1]
    if L[i][j] != 0:
        return .....
    for k in ....:
        L[i][j] = ....
        if .... :
            return ....
    L[i][j] = ....
    return ....
```

14. On suppose qu'au départ, il y a p cases déjà remplies.
Montrer qu'au maximum, la fonction `backtracking` est appelée 9^{81-p} fois.

15. Que renvoie la fonction `solution_sudoku(L)` si le sudoku L admet plusieurs solutions ? Et si L est le sudoku rempli de 0 ?

16. Dans l'algorithme précédent, on parcourt l'ensemble des cas dans l'ordre lexicographique. Comment améliorer celui-ci pour limiter le nombre d'appels à la variable `pos` ?

Problème n ° 2

Notations.
Pour m et n deux entiers naturels, $[\![m;n]\!]$ désigne l'ensemble des entiers k tels que $m \leqslant k \leqslant n$.

Ce problème a pour but d'étudier certains aspects de la géométrie algorithmique qui sont fortement utilisés en ingénierie et surtout en imagerie numérique. L'objectif est d'étudier deux algorithmes déterminant l'enveloppe convexe d'un ensemble de n points. Le premier, dit parcours de Jarvis*, s'exécute en temps $O(nN)$ où N est le nombre de sommets de l'enveloppe convexe. Le second, dit* balayage de Graham*, s'exécute en temps $O(n \ln n)$.*

Dans tout le problème, on se place dans un plan euclidien orienté muni d'un repère orthonormé direct (non visible sur les figures de ce problème).
On pourra utiliser les fonctions de la bibliothèque `math` supposée déjà importée.

On rappelle que la complexité (temporelle) d'un algorithme est le nombre d'opérations élémentaires (affectations, comparaisons, opérations arithmétiques) effectuées par un algorithme. Ce nombre s'exprime en fonction de la taille des données.

Un point P du plan muni d'un repère est représenté en Python par une liste de deux valeurs `P=[x,y]` où `x` et `y` sont deux nombres flottants correspondants aux coordonnées cartésiennes du point.
Un nuage de points est un ensemble $L = \{P_0, \ldots, P_{n-1}\}$ fini et non vide de points du plan. On le représente en Python par une liste `L` de longueur n, où pour tout entier i dans $[\![0; n-1]\!]$, `L[i]` représente le point P_i.

Partie A : Préliminaires

Calcul de la distance entre deux points du plan

La distance euclidienne entre deux points du plan P et Q de coordonnées respectives (x_P, y_P) et (x_Q, y_Q) est donnée par

$$PQ = \sqrt{(x_P - x_Q)^2 + (y_P - y_Q)^2}.$$

1. Écrire en Python une fonction `distance` prenant en arguments deux points P et Q du plan et renvoyant la valeur de la distance euclidienne entre ces deux points.

2. Un élève a écrit une fonction qui prend en argument un nuage de points de taille supérieure à 2 et qui détermine la distance minimale entre deux points de ce nuage. Voici le code de son programme en Python :

```
def distance_minimale(L):
    n=len(L)
    minimum=distance(L[0],L[1])
    i,j=0,0
    while i < n:
        while j < n:
            a=distance(L[i],L[j])
            if a<minimum:
                minimum=a
            j+=1
        i+=1
    return minimum
```

2.a Combien d'appels à la fonction `distance` sont effectués par cette fonction ?

2.b Quelle est la valeur renvoyée par cette fonction `distance_minimale` ?

2.c Corriger le programme de l'élève afin que la fonction `distance_minimale` soit correcte.

3. Écrire une fonction `distance_maximale` qui prend en argument un nuage de points L et qui renvoie la distance maximale entre deux points du nuage L en effectuant exactement $\dfrac{n(n-1)}{2}$ appels à la fonction `distance`. La fonction `distance_maximale` renverra également les indices d'un couple de points réalisant le maximum voulu. Par exemple :

```
>>> distance_maximale(L)
(0.9243140331952826, 1, 8)
```

Dans le nuage de points L donné en argument, la distance P_1P_8 est égale à 0.9243140331952826 qui est la distance maximale obtenue.

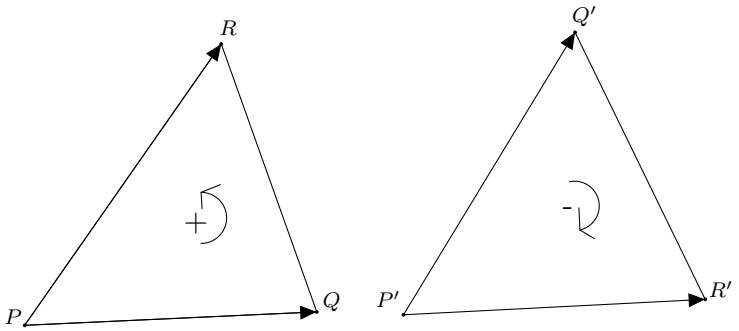
Recherche du point d'abscisse minimale

4. Écrire une fonction `point_abs_min` qui prend en paramètre un nuage de points L et qui renvoie l'indice du point de plus petite abscisse parmi les points du nuage de L. Si plusieurs points ont une abscisse minimale alors la fonction renverra parmi ces points, l'indice du point d'ordonnée minimale. Préciser la complexité temporelle de votre fonction lorsque le nuage est composé de n points.

Détermination de l'orientation de trois points du plan

Définition. Soient P, Q et R trois points du plan. On considère les vecteurs $\overrightarrow{PQ}$ et $\overrightarrow{PR}$ de coordonnées respectives $\begin{pmatrix} a \\ b \end{pmatrix}$ et $\begin{pmatrix} c \\ d \end{pmatrix}$. On note M la matrice $\begin{pmatrix} a & c \\ b & d \end{pmatrix}$.
On dit que l'orientation du triplet (P, Q, R)
– est en sens direct si le déterminant de la matrice M est strictement positif.
– est en sens indirect si le déterminant de la matrice M est strictement négatif.
– est un alignement si est seulement si le déterminant de la matrice M est nul.

Sur la figure ci-dessous, le triplet (P, Q, R) est en sens direct et le triplet (P', Q', R') est en sens indirect :



5. On suppose qu'un triplet (P, Q, R) est en sens direct. Quelle est l'orientation des triplets (Q, R, P) et (P, R, Q) ? Justifier votre réponse.

6. Écrire une fonction `orientation` qui prend en arguments 3 points du plan P, Q et R et qui renvoie 1 si le triplet (P, Q, R) est en sens direct, 0 si le triplet (P, Q, R) est un alignement, et -1 si le triplet (P, Q, R) est en sens indirect.

Étude de deux algorithmes de tri

7. la fonction `tri_bulle` ci-dessous prend en argument une liste `L` de nombres flottants et en effectue le tri en ordre croissant :

```
def tri_bulle(L):
    n=len(L)
    for i in range(n):
        for j in range(n-1,i,-1):
            if L[j]<L[j-1]:
                L[j],L[j-1]=L[j-1],L[j]        #échange d'éléments
```

7.a Lors de l'appel `tri_bulle(L)` où `L` est la liste `[5,2,3,1,4]`, donner le contenu de la liste `L` à la fin de chaque itération de la boucle `for i in range(n):`.

7.b On suppose que `L` est une liste non vide de nombres flottants. Montrer, pour tout $k \in [\![0; n]\!]$, la propriété $\mathcal{P}_k$:
« après k itérations de la première boucle, les k premiers éléments de la liste sont triés par ordre croissant et sont tous inférieurs aux $n - k$ éléments restants ».
En déduire que `tri_bulle(L)` trie bien la liste `L` en ordre croissant.

7.c Donner la complexité dans le meilleur des cas et dans le pire des cas de la fonction `tri_bulle`.

8 Soit la fonction `tri_fusion` suivante :

```
def tri_fusion(L):
    """ Fonction qui prend en argument une liste L de  nombres flottants
    et qui trie cette liste
    """
    n=len(L)
    if n<=1:
        return(L)
    else:
        m=n//2
        return (fusion(tri_fusion(L[0:m]),tri_fusion(L[m:n])))
```

8.a Écrire une fonction `fusion` qui prend en arguments deux listes triées `L1` et `L2` et qui renvoie une seule liste triée contenant les éléments de `L1` et `L2`.
La fonction `fusion` devra avoir une complexité en $O(n_1+n_2)$ où n_1 et n_2 sont les tailles respectives des listes `L1` et de `L2`.
Par exemple l'appel `fusion([2,4,7],[3,5,6,9])` renverra la liste `[2,3,4,5,6,7,9]`.

8.b On admet que la fonction `fusion` de la question précédente se termine. Montrer que la fonction `tri_fusion` se termine également.

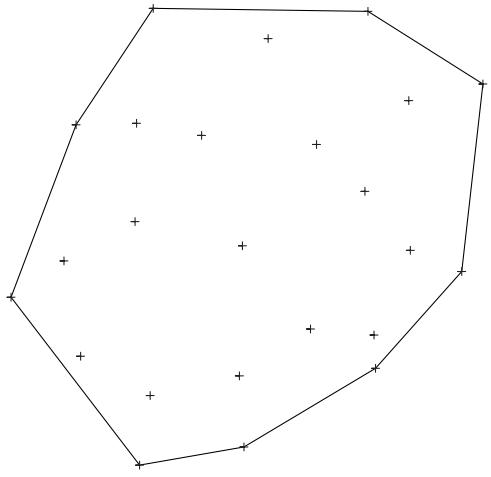
8.c On suppose que la longueur de la liste `L` est $n = 2^p$, où p est un entier naturel. Quelles sont alors les complexités dans le meilleur des cas et dans le pire des cas de `tri_fusion` ?

Partie B : Enveloppe convexe d'un nuage de points

Définition. Un ensemble S est **convexe** si pour tout couple de points (P, Q) de S, le segment $[PQ]$ est contenu dans S. L'**enveloppe convexe** d'un ensemble S est le plus petit ensemble convexe contenant S.

Dans cette partie, l'objet est d'étudier deux algorithmes permettant d'obtenir l'enveloppe convexe d'un ensemble fini de points du plan.

Théorème (admis)**.** Soit S un ensemble de n points du plan, avec $n > 1$. l'enveloppe convexe de S est constituée par un polygone P, sous-ensemble de S, des segments unissant les points successifs de P, et de tous les segments unissant deux points des segments précédents, comme l'illustre la figure ci-dessous.



Dans la suite du problème, on supposera que les nuages de points considérés ne contiennent pas 3 points distincts alignés. Cette hypothèse permettra de simplifier les algorithmes.

L'algorithme de Jarvis

La « marche de Jarvis » est un des algorithmes les plus naturels. Conçu en 1973, il consiste à reprendre l'image de l'emballage d'un cadeau.
La première idée de Jarvis pour déterminer l'enveloppe convexe est de chercher tout d'abord les segments qui forment ses côtés à l'aide de la propriété admise suivante :
Propriété : soit L un nuage de points. Pour tous points distincts P et Q de L, le segment $[PQ]$ est un côté de l'enveloppe convexe de L si les triplets (P, Q, R), où R est un point de L distinct de P et Q, ont tous la même orientation.

9. Voici une fonction « naïve » qui permet de déterminer les sommets de l'enveloppe convexe d'un nuage de points :

```
1  def jarvis(L):
2      """
3      Fonction qui reçoit en argument un nuage de points et qui renvoie
4      une liste contenant les indices des sommets de l'enveloppe
5      convexe de ce nuage
6      """
7      n=len(L)
8      EnvConvexe=[]
9      for i in range(n):
10         for j in range(n):
11             Listeorientation=[]
12             if i!=j:
13                 for k in range(n):
14                     if k!=i and k!=j:
15                             Listeorientation.append(orientation(L[i],L[j],L[k]))
16                 a=Listeorientation[0]
17                 sommet=True
18                 for v in Listeorientation:
19                     if (v!=a):
20                         sommet=False
21                 if  sommet and (i not in EnvConvexe) :
22                     EnvConvexe.append(i)
23                 if  sommet and (j not in EnvConvexe):
24                     EnvConvexe.append(j)
25     return EnvConvexe
```

9.a Expliquer à quoi servent les lignes 16 à 20 de cette fonction.

9.b Si on considère un nuage de n points avec $n \geqslant 3$, quelle est la complexité temporelle de cette fonction ? Justifier.

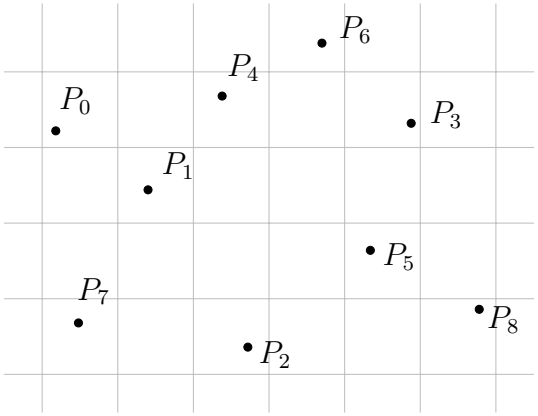
9.c Le script suivant trace-t-il le polygone formé par les sommets de l'enveloppe convexe du nuage L ? Justifier.

```
1  >>> import matplotlib.pyplot as plt
2  >>> Enveloppe=jarvis(L)
3  >>> plt.plot([L[i][0]  for i in Enveloppe],[L[i][1] for i in Enveloppe])
4  >>> plt.show()
```

Jarvis proposa ensuite une version plus efficace de son algorithme en voulant « ranger » les points du nuage autour d'un point de l'enveloppe. Il utilise une relation d'ordre sur les points du nuage.

Définition. Soit P un sommet de l'enveloppe convexe du nuage. Le point suivant sur l'enveloppe convexe est le plus petit pour la relation d'ordre $\preccurlyeq_P$ définie sur l'ensemble des sommets du nuage différents de P par :

$$Q \preccurlyeq_P R \Leftrightarrow \text{ le triplet } (P, Q, R) \text{ est en sens direct ou } Q = R.$$



Par exemple, dans le nuage points ci-dessus, on obtient le classement des points suivants pour la relation d'ordre $\preccurlyeq_{P_2}$:

$$P_8 \preccurlyeq_{P_2} P_5 \preccurlyeq_{P_2} P_3 \preccurlyeq_{P_2} P_6 \preccurlyeq_{P_2} P_4 \preccurlyeq_{P_2} P_1 \preccurlyeq_{P_2} P_0 \preccurlyeq_{P_2} P_7.$$

Le plus petit point pour la relation $\preccurlyeq_{P_2}$ est le point P_8.

10. Donner le classement des points du nuage de la figure ci-dessus pour la relation d'ordre $\preccurlyeq_{P_6}$.

Ainsi si on connaît un sommet P de l'enveloppe convexe d'un nuage de points alors le prochain point de l'enveloppe convexe à déterminer est celui qui est minimal pour la relation $\preccurlyeq_P$.

11. Écrire une fonction `prochain_sommet` qui prend en arguments un nuage de points L et l'indice d'un sommet P de ce nuage de points qui est un sommet de l'enveloppe convexe et qui renvoie l'indice du prochain sommet de l'enveloppe convexe. Cette fonction devra avoir une complexité en $O(n)$ où n est le nombre de points du nuage.

Comme le point du nuage d'abscisse minimale (et d'ordonnée minimale s'il y a plusieurs points d'abscisse minimale) fait partie de l'enveloppe convexe on peut construire un algorithme qui détermine au fur et à mesure tous les sommets de l'enveloppe convexe. On arrête l'algorithme quand la fonction `prochain_sommet` renvoie l'indice du sommet de départ.

12. Recopier et compléter la fonction suivante afin qu'elle renvoie l'enveloppe convexe d'un nuage de points L.

```
def jarvis2(L):
    i=point_abs_min(L)
    suivant=prochain_sommet(L,i)
    Enveloppe=[i,suivant]   #initialisation de la liste des indices des sommets de
    l'enveloppe convexe
    while (.......) :
        ......
        ......
    return Enveloppe
```

13. Soit L un nuage de n points dont on sait que l'enveloppe convexe est un polygone à N sommets. Montrer que l'algorithme décrit par la fonction `jarvis2` possède une complexité en $O(n \times N)$.

L'algorithme de Graham - Andrew

En 1972, Graham et Andrew proposèrent une méthode pour déterminer l'enveloppe convexe d'un nuage de points. Leur algorithme est basé sur une méthode de tri des points.

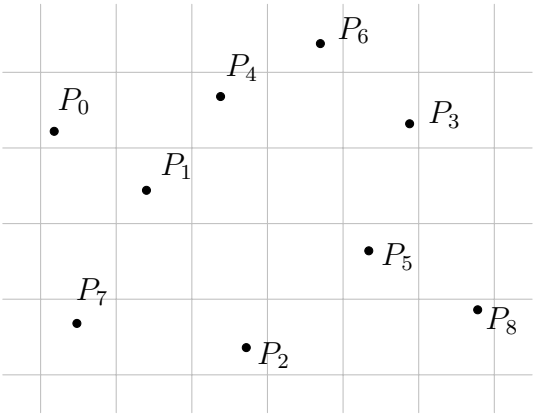
La première étape de l'algorithme de Graham et Andrew est de trier les n points du nuage L par ordre croissant d'abscisses (si deux points ont la même abscisse on classera ces points suivant leurs ordonnées). On supposera donnée une fonction `tri_nuage` prenant en entrée un nuage de points `L` et réalisant cette opération en complexité $O(n \log n)$ où n est le nombre de points du nuage. ($n \geqslant 3$).

L'idée de l'algorithme est de balayer le nuage de points dans l'ordre croissant de leurs abscisses tout en mettant à jour l'enveloppe convexe des points. L'enveloppe convexe a été scindée en deux listes `EnvSup` et `EnvInf`.
Dans ces listes `EnvSup` et `EnvInf`, on maintient l'enveloppe convexe des points déjà traités. Chaque nouvel indice du point P du nuage est ajouté à `EnvSup` et `EnvInf`, puis tant que l'avant dernier-point de `EnvSup` rend la séquence non-convexe, il est enlevé (de même pour `EnvInf`).

14. Voici en Python la fonction qui permet d'obtenir l'enveloppe convexe :

```
def graham_andrew(L):
    L=tri_nuage(L)
    EnvSup=[]
    EnvInf=[]
    for i in range(len(L)):
        while len(EnvSup)>=2 and orientation(L[i],L[EnvSup[-1]],L[EnvSup[-2]])<=0:
            EnvSup.pop()
        EnvSup.append(i)
        while len(EnvInf)>=2 and orientation(L[EnvInf[-2]],L[EnvInf[-1]],L[i])<=0:
            EnvInf.pop()
        EnvInf.append(i)
    return EnvInf[:-1]+EnvSup[::-1]
```

14.a A partir du nuage de points représentés ci-dessous, donner le contenu de la liste `EnvSup` à chaque itération de la boucle for.
Donner également le résultat de `orientation(P,EnvSup[-1],EnvSup[-2])<=0`.



14.b Montrer la terminaison de la fonction `graham_andrew`.

14.c Montrer que la complexité de `graham_andrew` est en $O(n \log n)$ où n est la taille du nuage de points.

Annexe

Langage Python

Listes

```
>>> maListe = [1,8,'e']    # définition d'un liste
>>> maListe[0]             # le premier élément d'une liste a l'indice 0
1
>>> maListe[1]
8
>>> maListe[-1]            # le dernier élément de la liste
'e'
>>> maListe[-2]            # l'avant dernier élément de la liste
8
>>> len(maListe)           # longueur d'une liste
3
>>> maListe.append(12)     # ajout d'un élément en fin de liste
>>> maListe
[1, 8, 'e', 12]
>>> maListe.remove(8)      # suppression du premier élément égale à 8
>>> maListe
[1, 'e', 12]
>>> maListe.pop()          # retourne le dernier élément et le supprime de la liste
'e'
>>> maListe
[1, 12]
>>> maListe.insert(1,'a') # insert l'élément 'a' à l'indice 1 du tableau
>>> maListe
[1, 'a', 12]
>>>range(len(maListe)) :  # parcours des indices de la liste
>>> range(8)               # parcours des indices entiers de 0 à 8 exclus
0 1 2 3 4 5 6 7
>>>range(3,8)             # parcours des indices entiers de 3 inclus à 8 exclus
3 4 5 6 7
>>> range(3,8,2)          # parcours des indices entiers de 3 inclus
3 5 7                     # à 8 exclus avec un pas de 2
>>>A= [1,'e',8]+ [2,'tu'] # concaténation de deux listes
[1,'e',8,2,'tu']
>>>A[1:4:2]               # renvoie la liste des éléments de A d'indice 1 inclus
['e',2]                   #  à 4 exclus avec un pas de 2.
>>>A[:-1]                 # renvoie la liste des éléments de A à l'exclusion
[1,'e',8,2]               # du dernier
>>>A[::-1]                #renvoie la liste des éléments de A, du dernier au premier
['tu',2,8,'e',1]
```

Complexité de certaines procédures en langage Python

`Liste.pop()`	$O(1)$
`Liste.append(element)`	$O(1)$
`element in Liste`	$O(\text{len(Liste)})$
`element not in Liste`	$O(\text{len(Liste)}$

Module `math`

```
>>> from math import *    # chargement du module math

>>> sqrt(2)               # racine de 2
1.4142135623730951
```