TP N°01 :
REGLES DE PROGRAMMATION, ASSERTION

OBJECTIFS DU TP

- Appliquer les règles de programmation
- Se familiariser avec la manipulation de listes de listes (ou tableau)
- Etudier la complexité d'une fonction.

Document 1 : Complexité et opérations élémentaires

La complexité d'un algorithme est une mesure de la quantité de ressources nécessaires pour mener à bien son exécution.
On peut considérer deux mesures différentes de la complexité :

- la **complexité temporelle** (complexité en temps) est une mesure de l'évolution du temps de calcul nécessaire à l'exécution d'un algorithme en fonction de la **taille d'instance**;
- la **complexité spatiale** (complexité en espace) est une mesure de l'évolution du nombre de cases mémoire requises lors de l'exécution d'un algorithme en fonction de la taille d'instance.

Dans ce TP, on s'intéressera uniquement à la complexité temporelle qui dépend du nombre d'opérations élémentaires requises pour mener à bien l'algorithme et du temps nécessaire à la machine utilisée pour réaliser **une** opération élémentaire. Pour s'affranchir de la machine utilisée, on dénombrera seulement le nombre d'opérations élémentaires.

On considérera comme une **opération élémentaire** chacun des éléments suivants :

- l'affectation d'une variable (« = »)
- l'appel aux méthodes natives de Python (print, abs, etc.) ou l'utilisation de mots-clés (return, break, etc.)
- les opérations arithmétiques élémentaires sur les entiers ou les flottants : « + », « - », « * », « / », « // » pour le quotient et « % » pour le reste, (« ** »), etc.
- les opérations de logique : appartenance (« in »), coïncidence (« is »), conjonction (« and »), disjonction (« or »), négation (« not »), etc.
- Les opérations de comparaison : égalité (« == »), différence (« != »), supériorité (« > » et « >= »), infériorité (« < » et « <= ») etc.

Exemple :

```
def somme(L :[int])-> int :
      """ fonction qui calcul la somme des termes d'une liste de nombres """
      S = 0                          # 1 affectation
      for k in range (len(L)) :      # utilisation de len, for, in range
            S = S + L[k]             # 1 affectation + 1 somme
      return S                       # 1 return
```

Le calcul de `somme([1,2,3])` *nécessite, 3 itérations de la boucle* `for`*, donc 11 opérations élémentaires. En généralisant pour une liste de taille n, on aurait* $5 + 2 \times n$ *opérations. Etant donné que n est souvent très élevé, on pourra considérer un ordre de grandeur de la complexité, soit ici n opérations, on parle de complexité linéaire.*

Pour ce TP, vous aurez besoin de fonctions des bibliothèques `numpy` et `matplotlib.pyplot`.

Exercice 1 : Analyse de code

Le code ci-dessous semble correct et pourtant renvoie un message d'erreur lorsqu'on l'exécute.

```
def f(L):
      s = 0
      for t in L:
            s += t
      L. append (s)

def g(L):
      Res = f(L)
      return Res

L = [1 ,2 ,3]
Res = g(L)
```

Q1. Sans l'implémenter dans votre console Python expliquez pourquoi le code ne renvoie pas ce que l'on croit et proposez une modification.

Q2. Compléter ces fonctions avec la signature de type et une documentation.

Q3. Déterminer l'ordre de grandeur du nombre d'opérations élémentaires réalisées par les fonctions `f` et `g` en fonction du nombre n d'éléments dans la liste L (on parle de taille d'instance).

Exercice 2 : Sans doublon

Q1. Ecrire une fonction `SansDoublons` qui prend en entrée une liste d'éléments L et renvoie une nouvelle liste contenant exactement une seule fois chaque élément de L.

Deux contraintes :

- la liste L ne doit être parcourue qu'une seule fois,
- L ne doit pas être modifiée par la fonction.

Q2. Ajouter une commande d'assertion qui vérifie que la variable d'entrée est bien une liste.

Q3. Pour une liste L de taille n fixée, dans quel cas aurait-on la pire complexité en temps ? La meilleure complexité en temps ?

Rq : il est également possible de s'intéresser à la complexité moyenne, mais cela devient rapidement compliqué. En CPGE, on se concentrera uniquement sur la complexité dans le pire des cas pour comparer les algorithmes.

Q4. Déterminer l'ordre de grandeur du nombre d'opérations élémentaires réalisées par la fonction `SansDoublons` en fonction du nombre n d'éléments dans la liste.

Exercice 3 : Où sera le lac ?

On se trouve dans une vallée où l'altitude au point de coordonnées entières (x ; y) est stockée dans un tableau 2D appelé `alt`. Ainsi, si `alt[3][8]=10` cela signifie qu'au point de coordonnées (3 ; 8), l'altitude est de 10 mètres.
On souhaite trouver un endroit où l'eau formera un petit lac quand il se mettra à pleuvoir. On cherche donc des coordonnées (x; y) telles que l'altitude en (x; y) soit plus petite que celle de ses 8 cases voisines (x-1; y-1), (x; y - 1), (x + 1; y - 1) .. etc.
On suppose que l'on sait que l'on est dans une vallée creuse et qu'un tel point existe. L'algorithme que nous allons utiliser consistera à partir d'un point de la carte, et à avancer tant qu'on peut à chaque pas vers une case voisine d'altitude strictement inferieure à l'altitude de la case où l'on se trouve.

Q1. Créer une fonction `altitude` prenant en argument deux entiers A et B et renvoyant un tableau de A lignes et B colonnes tel que :

$$\forall i \in [\![0, A]\!] \quad et \; \forall j \in [\![0, B]\!] \quad , alt[i][j] = i * i + i * j - 60 * i + 1.2 * j * j - 90 * j + 2180$$

Q2. Ecrire une liste d'instruction permettant d'obtenir un tableau des altitudes `alt` de 100 par 100.

Q3. Ecrire une fonction `deplace` qui prend en argument un tableau `alt` contenant les altitudes et deux entiers i et j représentant les coordonnées (i, j) d'un point sur la carte. Cette fonction renvoie :

- Une liste [newI,newJ] contenant les coordonnées d'UNE case voisine de (i, j) dont l'altitude est strictement inférieure à celle de la case (i, j) lorsque c'est possible ;
- -1 si cela n'est pas possible, c'est-à-dire si la case (i; j) est d'altitude minimale parmi ses voisines.

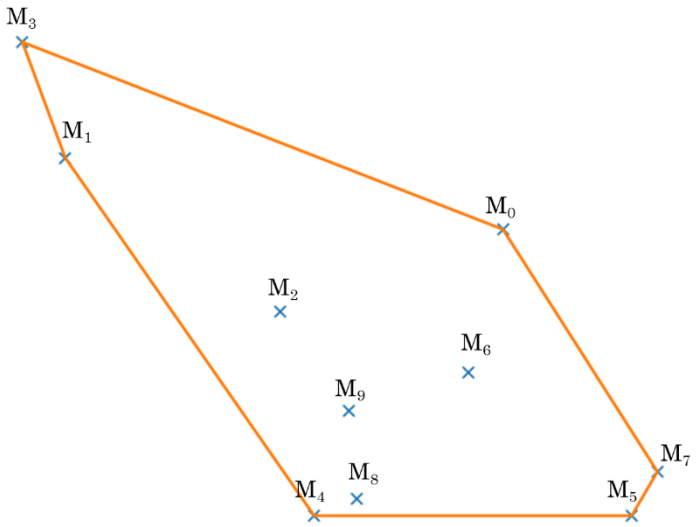
Q4. Ecrire une fonction `futur_lac` reprenant l'algorithme proposé dans l'énoncer pour trouver l'emplacement du futur lac en partant d'une carte `alt` et du point de coordonnées (Di ;Dj),

Q5. Trouver l'emplacement d'un futur lac sur la carte `alt`.

Exercice 4 : Enveloppe convexe d'une famille de points

Nous considérons une famille $(M_i)_{0\leq i<n}$ de points distincts d'un plan dans le repère orthonormé direct $(O,\vec{u},\vec{v})$. Chaque point est représenté par un tuple de coordonnées (x_i, y_i). La famille de points$(M_i)_{0\leq i<n}$ est représentée par une liste $L = [(x_0, y_0), \dots, (x_{n-1}, y_{n-1})]$.

L'enveloppe convexe $\mathcal{C}$ de cette famille est le plus petit polygone convexe contenant tous les points M_i. Le but de cet exercice est de calculer la liste E dite « bien ordonnée » des sommets de $\mathcal{C}$, c'est-à-dire la liste des sommets $\mathcal{S}$ parcourus dans le sens trigonométrique direct, en commençant par le point M_{i0} situé le plus bas parmi ceux situés le plus à gauche. Ainsi avec les points représentés sur la figure ci-dessous, $i_0 = 3$ et nous cherchons à construire la liste E = $[M_3, M_1, M_4, M_5, M_7, M_0]$.



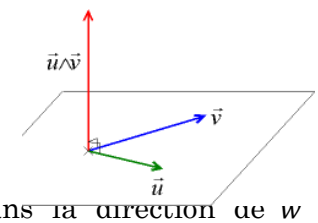
Les algorithmes d'enveloppe convexe sont largement utilisés en infographie et en vision. Ils sont impliqués dans le rendu, l'animation, la détection de collision, la reconnaissance de forme et le traitement d'images. Par exemple, les algorithmes d'enveloppe convexe peuvent aider à créer des ombres et des reflets réalistes en trouvant la silhouette d'un objet à partir d'une source de lumière ou d'un point de vue. Ils peuvent également aider à simplifier les modèles complexes en trouvant la plus petite forme convexe qui s'en rapproche. De plus, les algorithmes d'enveloppe convexe peuvent aider à identifier et à classer les objets dans les images en trouvant leurs contours convexes et en les comparant à des modèles ou des entités prédéfinis.

Document 2 : Aide / rappel : produit vectoriel

Le **produit vectoriel**, noté ∧ entre deux vecteurs $\vec{u}$ *et* $\vec{v}$ est défini par :

Soit $\vec{u}$ *et* $\vec{v}$ deux vecteurs de coordonnées respectives dans $\mathbb{R}^3$ $\begin{pmatrix} x_u \\ y_u \\ z_u \end{pmatrix}$ et $\begin{pmatrix} x_v \\ y_v \\ z_v \end{pmatrix}$.

$\vec{w} = \vec{u} \wedge \vec{v}$, de coordonnées $\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} y_u.z_v - z_u.y_v \\ z_u.x_v - x_u.z_v \\ x_u.y_v - y_u.x_v \end{pmatrix}$



$\vec{w} = \vec{u} \wedge \vec{v} = ||\vec{u}||.||\vec{v}||.\sin(\widehat{\vec{u},\vec{v}})\ \overrightarrow{e_w}$ $\overrightarrow{e_w}$ étant le vecteur unitaire dans la direction de w perpendiculaire à $\vec{u}$ *et* $\vec{v}$.

Pour s'épargner le traitement de cas particuliers, nous supposerons que trois points quelconques de la famille $(M_i)_{0\leq i<n}$ ne sont jamais alignés.

Si A, B et C sont trois points non alignés du plan, nous dirons que « (A, B, C) tourne gauche » si l'angle entre $\overrightarrow{AB}$ et $\overrightarrow{AC}$ admet une mesure entre $]0,\pi[$.

Q1. Ecrire une fonction `tourne_gauche` qui prend en argument trois points A, B et C et qui renvoie le booléen `True` si (A, B, C) tourne gauche et `False` sinon.

Q2. Ecrire une fonction `point_depart` qui, à partir d'une liste L de coordonnées de points renvoie l'indice i_0 du point de départ du tracé de l'enveloppe convexe.

Q3. Ecrire une fonction `tri_sommets` qui, appliquées à L et i_0, renvoie la pile* P des points M_i pour $i \neq i_0$, triée dans l'ordre décroissant des angles que font les vecteurs $\vec{u}$ et $\overrightarrow{M_{i0}M_i}$. On utilisera la fonction `tourne_gauche`.

**Une pile est une liste dans laquelle seul le dernier élément est accessible (comme une pile de crêpe). La résolution de ce problème est proche de celle du jeu des tours d'Hanoï illustré ci-contre.*

Dans l'exemple, on aurait :

P = $[M_3, M_0, M_2, M_7, M_6, M_5, M_9, M_8, M_4, M_1]$

Q4. Ecrire une fonction `enveloppe_convexe` qui, appliquée à L, renvoie la liste E en utilisant la méthode suivante :

a. On calcule i_0 puis P

b. On initialise une pile E qui ne contient que M_{i0} au début du calcul

c. On vide P en modifiant E de sorte que chaque tour de boucle, E soit la liste bien ordonnée des sommets de l'enveloppe convexe des points de L qui ne sont pas présents dans P.

Q5. Déterminer l'ordre de grandeur du nombre d'opération (complexité) de l'étape (c) en fonction du nombre de points n. Puis déterminer la complexité de la fonction complète `enveloppe_convexe`. Peut-on améliorer cette complexité ?

Q6. Ecrire une fonction `tracer` qui prend en argument un entier n, simule le tirage de points de coordonnées (x_i, y_i), et trace le nuage de ces points ainsi que le contour de leur enveloppe convexe.