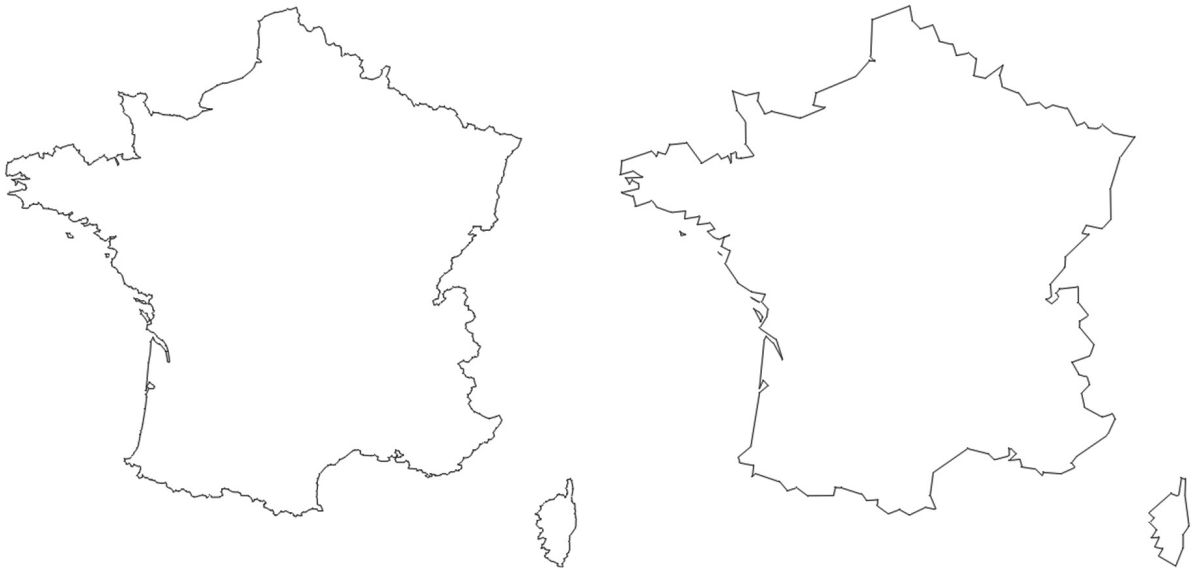
Exercices sur diviser pour régner

Exercice 1 : *Cet exercice porte sur la programmation en Python, la récursivité et la méthode "diviser pour régner".*

Une ligne polygonale est constituée d'une liste ordonnée de points, appelés sommets, joints par des segments. L'algorithme de Douglas-Peuker permet de simplifier une ligne polygonale en supprimant certains de ses sommets. L'effet de l'algorithme appliqué aux lignes polygonales du contour de la France métropolitaine est illustré ci-dessous.



Avant application de l'algorithme Après application de l'algorithme

On implémentera cet algorithme dans la dernière question de l'exercice. Pour cela nous allons d'abord implémenter des fonctions auxiliaires.

On suppose dans la suite que les sommets sont des points du plan dont les coordonnées (x,y) dans un repère orthonormé fixé sont représentées par des tuples de longueur 2.

1) La distance qui sépare deux points A et B de coordonnées (x_A,y_A) et (x_B,y_B) donnée par la formule $\sqrt{(x_B-x_A)^2+(y_B-y_A)^2}$.
   On rappelle que la fonction `sqrt` du module `math` de Python renvoie la racine carrée d'un nombre positif ou nul.
   a) Écrire une instruction qui permet d'importer la fonction `sqrt` du module `math`.
   b) Supposant l'import réalisé, écrire une fonction `distance_points(a,b)` qui prend en argument deux tuples `a` et `b` représentant les coordonnées de deux points et renvoie la distance qui les sépare.
2) On dispose d'une fonction `distance_point_droite(p, a, b)` qui prend en argument les tuples représentant les coordonnées respectives des points P, A et B, et qui renvoie la distance du point P à la droite (AB). L'exécution de cette fonction produit une erreur dans le cas où les points A et B sont égaux.
   À l'aide des fonctions `distance_points` et `distance_point_droite`, écrire une fonction `distance(p, a, b)` qui renvoie la distance entre le point P et la droite (AB) si les points A et B sont distincts et la distance AP sinon.

Dans la suite, on dira que la fonction `distance` calcule la distance entre le point P et les points A et B, éventuellement confondus.

3) On a besoin d'une fonction `le_plus_loin(ligne)` qui prend en argument une liste de tuples représentant les coordonnées des points d'une ligne polygonale.

Cette fonction doit renvoyer un tuple composé de :

- l'indice du point de coordonnées `p` de la ligne polygonale d'extrémités `deb` et `fin`, pour lequel la distance `distance(p, deb, fin)` est la plus grande ;
- la valeur correspondante de cette distance.

On fournit le code incomplet suivant :

```
def le_plus_loin(ligne):
    n = len(ligne)
    deb = ligne[0]
    fin = ligne[n-1]
    dmax = 0
    indice_max = 0
    for idx in range(1, n-1):
        p = ...
        d = distance(p, deb, fin)
        if ...:
            ...
            ...
    return ...
```

Recopier et compléter le code de cette fonction.

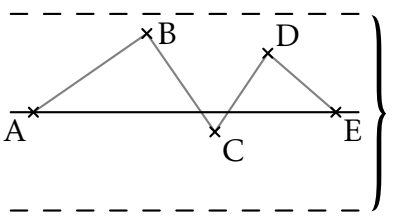
4) Écrire une fonction `extrait(tab, i, j)` qui renvoie la copie du tableau `tab` des cases d'indice `i` inclus à `j` inclus pour $0 \leq i \leq j < $ `len(tab)`.

L'appel `extrait([7, 4, 9, 12], 2, 3)` renverra ainsi `[9, 12]`.

L'algorithme de Douglas-Peuker repose sur une stratégie de type "diviser pour régner". Pour éliminer des sommets "proches de l'alignement", un seuil est fixé.

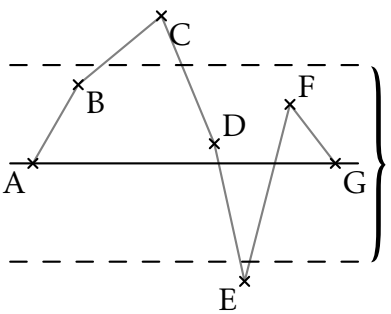
Étant donnée une ligne polygonale, le principe de l'algorithme est le suivant :

- si la ligne ne contient qu'un ou deux sommets, l'algorithme se termine ;
- sinon, on considère la droite formée par les extrémités de la ligne (son premier et dernier sommet), et on sélectionne le point le plus éloigné de cette droite (dans le cas où les extrémités sont confondues, on sélectionne le point le plus éloigné de celles-ci) :
  - si la distance entre le point sélectionné et la droite (ou les extrémités lorsqu'elles sont confondues) est inférieure au seuil fixé, on ne conserve que les extrémités de la ligne polygonale ;



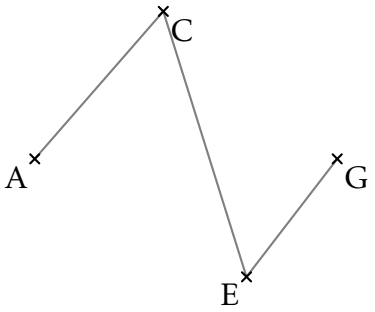
Distances inférieures au seuil
on ne conserve que les points A et E

  - sinon, on applique l'algorithme de manière récursive aux deux parties de la ligne polygonale formées de la séquence formée du premier sommet jusqu'au sommet sélectionné d'une part et de la séquence formée du sommet sélectionné jusqu'au dernier sommet d'autre part. L'algorithme renvoie alors la concaténation des séquences simplifiées ainsi obtenues.



seuil

L'algorithme appelé sur la ligne polygonale [A,B,C,D,E,F,G] ci-dessus, va récursivement être appelé sur les lignes polygonales [A,B,C] et [C,D,E,F,G]. La ligne polygonale que l'on obtiendra à la fin de l'algorithme sera [A,C,E,G].

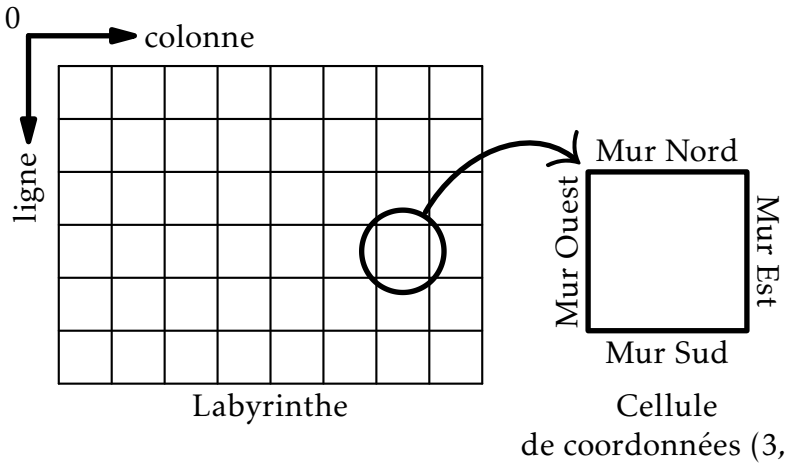


5) L'algorithme de Douglas-Peuker est implémenté par la fonction `simplifie` ci-dessous, qui prend en argument la ligne polygonale et le seuil choisi.

```
def simplifie(ligne, seuil):
    n = len(ligne)
    if n <= 2:
        return ...
    else:
        indice_max, dmax = le_plus_loin(ligne)
        if dmax <= seuil:
            return ...
        else:
                    } # bloc à écrire
```

Recopier et compléter le code de cette fonction.

Exercice 2 : *Cet exercice aborde la programmation objet et la méthode diviser pour régner.*



Un labyrinthe est composé de cellules possédant chacune quatre murs (voir ci-dessus). La cellule en haut à gauche du labyrinthe est de coordonnées `(0, 0)`. On définit la classe `Cellule` ci-dessous. Le constructeur possède un attribut `murs` de type `dict` dont les clés sont `'N'`, `'E'`, `'S'` et `'O'` et dont les valeurs sont des booléens (`True` si le mur est présent et `False` sinon).

```
class Cellule:
    def __init__(self, murNord, murEst, murSud, murOuest):
        self.murs = {'N':murNord, 'E':murEst, 'S':murSud, 'O':murOuest}
```

1) Compléter l'instruction Python suivante permettant de créer une instance `cellule` de la classe `Cellule` possédant tous ses murs sauf le mur Est.

```
cellule = Cellule(.....................................)
```

2) Le constructeur de la classe `Labyrinthe` ci-dessous possède un seul attribut `grille`. La méthode `construire_grille` permet de construire un tableau à deux dimensions `hauteur` et `longueur` contenant des cellules possédant chacune ses quatre murs. Compléter les lignes 7 à 11 de la classe `Labyrinthe`.

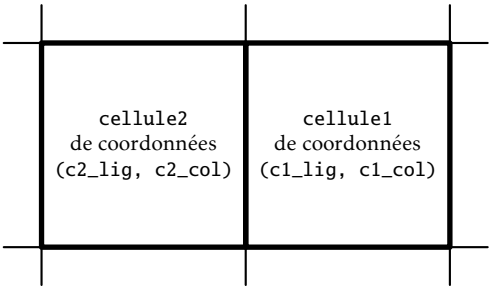
```
1  class Labyrinthe:
2      def __init__(self, hauteur, longueur):
3          self.grille=self.construire_grille(hauteur, longueur)
4
5      def construire_grille(self, hauteur, longueur):
6          grille = []
7          for i in range(.................):
8              ligne = []
9              for j in range(.................):
10                 cellule = ...
11                 ligne.append(.................)
12             grille.append(ligne)
13         return grille
```

Pour générer un labyrinthe, on munit la classe `Labyrinthe` d'une méthode `creer_passage` permettant de supprimer des murs entre deux cellules ayant un côté commun afin de créer un passage. Cette méthode prend en paramètres les coordonnées `(c1_lig, c1_col)` d'une cellule notée `cellule1` et les coordonnées `(c2_lig, c2_col)` d'une cellule notée `cellule2` et crée un passage entre `cellule1` et `cellule2`.

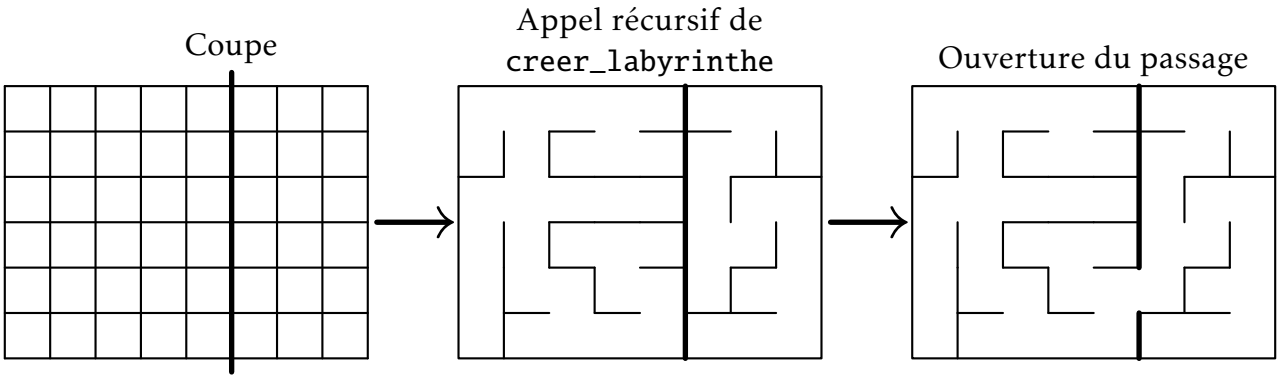
```
15     def creer_passage(self, c1_lig, c1_col, c2_lig, c2_col):
16         cellule1 = self.grille[c1_lig][c1_col]
17         cellule2 = self.grille[c2_lig][c2_col]
18         # cellule2 au Nord de cellule1
19         if c1_lig - c2_lig == 1 and c1_col == c2_col:
20             cellule1.murs['N'] = False
21             ....
22         # cellule2 à l'Ouest de cellule1
23         elif ..............................................:
24             ....
25             ....
```

3) La ligne 20 permet de supprimer le mur Nord de `cellule1`. Un mur de `cellule2` doit aussi être supprimé pour libérer un passage entre `cellule1` et `cellule2`. Écrire l'instruction Python que l'on doit ajouter à la ligne 21.

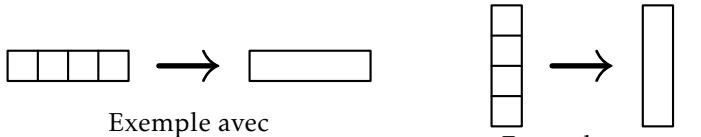
4) Compléter le code Python des lignes 23 à 25 qui permettent le traitement du cas où `cellule2` est à l'Ouest de `cellule1` :



Pour créer un labyrinthe, on utilise la méthode diviser pour régner en appliquant récursivement l'algorithme `creer_labyrinthe` sur des sous-grilles obtenues en coupant la grille en deux puis en reliant les deux sous-labyrinthes en créant un passage entre eux.



La méthode `creer_labyrinthe` permet, à partir d'une grille, de créer un labyrinthe de hauteur `haut` et de longueur `long` dont la cellule en haut à gauche est de coordonnées `(ligne, colonne)`.
Le cas de base correspond à la situation où la grille est de hauteur 1 ou de largeur 1. Il suffit alors de supprimer tous les murs intérieurs de la grille.



Exemple avec
`haut` = 1 et `long` = 4

Exemple avec
`haut` = 4 et `long` = 1

5) Compléter les lignes 28 à 33 de la méthode `creer_labyrinthe` traitant le cas de base.

```
27      def creer_labyrinthe(self, ligne, colonne, haut, long):
28          if haut == 1 : # Cas de base
29              for k in range(.....................):
30                  self.creer_passage(ligne, colonne+k, ligne, colonne+k+1)
31          elif long == 1: # Cas de base
32              for k in range(.....................):
33                  self.creer_passage(.....................................)
34          else: # Appels récursifs
35              # Code non étudié (Ne pas compléter)
```

6) Dans cette question, on considère une grille de hauteur `haut` = 4 et de longueur `long` = 8 dont chaque cellule possède tous ses murs.
On fixe les deux contraintes supplémentaires suivantes sur la méthode `creer_labyrinthe` :

- Si `haut` ≥ `long`, on coupe horizontalement la grille en deux sous-labyrinthes de même dimension.
- Si `haut` < `long`, on coupe verticalement la grille en deux sous-labyrinthes de même dimension.

L'ouverture du passage entre les deux sous-labyrinthes se fait le plus au Nord pour une coupe verticale et le plus à l'Ouest pour une coupe horizontale.

Dessiner le labyrinthe obtenu suite à l'exécution complète de l'algorithme `creer_labyrinthe` sur cette grille.