Exercices - Diviser pour régner

EXERCICE 1. : SUJET 2023

L'objectif de cet exercice est de trouver les deux points les plus proches dans un nuage de points pour lesquels on connaît les coordonnées dans un repère orthogonal.

On rappelle que la distance entre deux points A et B de coordonnées (xA ; yA) et (xB ; yB) est donnée par la formule $AB = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$.

Les coordonnées d'un point seront stockées dans un tuple de deux nombres réels. Le nuage de points sera représenté en Python par une liste de tuples de taille n, n étant le nombre total de points. On suppose qu'il n'y a pas de points confondus (mêmes abscisses et mêmes ordonnées) et qu'il y a au moins deux points dans le nuage. Pour calculer la racine carrée, on utilisera la fonction *sqrt* du module *math.*

1- Cette partie comprend plusieurs questions générales :

a) Expliquer le résultat suivant :

```
>>> 0.1 + 0.2 == 0.3
False
```

b) Expliquer l'erreur suivante :

```
>>> point_A = (3, 4)
>>> point_A[0]
3
>>> point_A[0] = 2
Traceback (most recent call last):
    File "<console>", line 1, in <module>
TypeError : 'tuple' object does not support item assignment
```

2- On définit la classe Segment ci-dessous :

```
1 from math import sqrt
2 class Segment:
3     def __init__(self, point1, point2):
4         self.p1 =  point1
5         self.p2 = point2
6         self.longueur = ..............  # à compléter
```

a) Recopier et compléter la ligne 6 du constructeur de la classe Segment.

La fonction *liste_segments* donnée ci-dessous prend en paramètre une liste de points et renvoie une liste contenant des objets *Segment* qu'il est possible de construire à partir de ces points. On considère les segments [AB] et [BA] comme étant confondus et ajoutera un seul objet dans la liste.

```
1 def liste_segments(liste_points):
2     n = len(liste_points)
3     segments = []
4     for i in range(..............):
5         for j in range(.............., n):
6             # On construit le segment à partir des points i et j.
7             seg =  ..............................................
8             segments.append(seg)  # On l'ajoute à la liste
9     return segments
```

b) Recopier la fonction sans les commentaires et compléter le code manquant.

c) Donner en fonction de *n* la longueur de la liste segments. Le résultat peut être laissé sous la forme d'une somme.

d) Donner, en fonction de *n*, la complexité en temps de la fonction *liste_segments*.

3- L'objectif de cette partie est d'écrire la fonction de recherche des deux points les plus proches en utilisant la méthode diviser-pour-régner.

On dispose de deux fonctions : moitie_gauche (respectivement moitie_droite) qui prennent en paramètre une liste et qui renvoient chacune une nouvelle liste contenant la moitié gauche (respectivement la moitié droite) de la liste de départ. Si le nombre d'éléments de celle-ci est impair, l'élément du centre se trouve dans la partie gauche.
Exemples :

```
>>> liste = [1, 2, 3, 4]
>>> moitie_gauche(liste)
[1, 2]
>>> moitie_droite(liste)
[3, 4]
```

```
>>> liste = [1, 2, 3, 4, 5]
>>> moitie_gauche(liste)
[1, 2, 3]
>>> moitie_droite(liste)
[4, 5]
```

Question : Écrire la fonction *plus_court_segment* qui prend en paramètre une liste d'objets Segment et renvoie l'objet Segment dont la longueur est la plus petite.
On procédera de la façon suivante :

- Tester si le cas de base est atteint, c'est-à-dire lorsque la liste contient un seul segment ;
- Découper la liste en deux listes de tailles égales (à une unité près) ;
- Appeler récursivement la fonction pour rechercher le minimum dans chacune des deux listes ;
- Comparer les deux valeurs récupérées et renvoyer la plus petite des deux.

4. On considère les trois points A(3 ; 4), B(2 ; 3) et C(−3 ; −1).

a) Donner l'instruction Python permettant de construire la variable nuage_points contenant les trois points A, B et C.

b) En utilisant les fonctions de l'exercice, écrire les instructions Python qui affichent les coordonnées des deux points les plus proches du nuage de points nuage_points.

EXERCICE 2. : SUJET 2021

1. **a.** Quel est l'ordre de grandeur du coût, en nombre de comparaisons, de l'algorithme de tri fusion pour une liste de longueur n ?

   b. Citer le nom d'un autre algorithme de tri. Donner l'ordre de grandeur de son coût, en nombre de comparaisons, pour une liste de longueur n. Comparer ce coût à celui du tri fusion. Aucune justification n'est attendue.

L'algorithme de tri fusion utilise deux fonctions `moitie_gauche` et `moitie_droite` qui prennent en argument une liste `L` et renvoient respectivement :

- la sous-liste de `L` formée des éléments d'indice strictement inférieur à `len(L)//2` ;
- la sous-liste de `L` formée des éléments d'indice supérieur ou égal à `len(L)//2`.

On rappelle que la syntaxe `a//b` désigne la division entière de `a` par `b`.

Par exemple,

```
>>> L = [3, 5, 2, 7, 1, 9, 0]
>>> moitie_gauche(L)
[3, 5, 2]
>>> moitie_droite(L)
[7, 1, 9, 0]
```

```
>>> M = [4, 1, 11, 7]
>>> moitie_gauche(M)
[4, 1]
>>> moitie_droite(M)
[11, 7]
```

L'algorithme utilise aussi une fonction `fusion` qui prend en argument deux listes triées `L1` et `L2` et renvoie une liste `L` triée et composée des éléments de `L1` et `L2`.

On donne ci-dessous le code python d'une fonction récursive `tri_fusion` qui prend en argument une liste `L` et renvoie une nouvelle liste triée formée des éléments de `L`.

```
def tri_fusion(L):
  n = len(L)
  if n<=1 :
     return L
  print(L)
  mg = moitie_gauche(L)
  md = moitie_droite(L)
  L1 = tri_fusion(mg)
  L2 = tri_fusion(md)
  return fusion(L1, L2)
```

2. Donner la liste des affichages produits par l'appel suivant.

```
tri_fusion([7, 4, 2, 1, 8, 5, 6, 3])
```

On s'intéresse désormais à différentes fonctions appelées par `tri_fusion`, à savoir `moitie_droite` et `fusion`.

3. Écrire la fonction `moitie_droite`.

4. On donne ci-dessous une version incomplète de la fonction `fusion`.

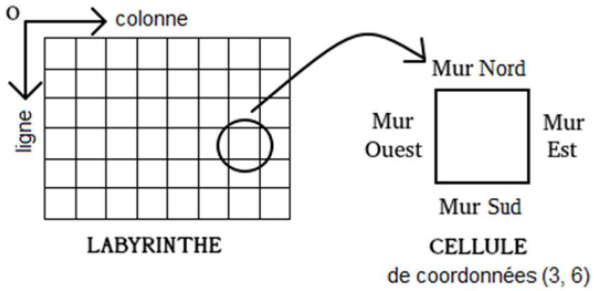
```
1.  def fusion(L1, L2):
2.      L = []
3.      n1 = len(L1)
4.      n2 = len(L2)
5.      i1 = 0
6.      i2 = 0
7.      while i1 < n1 or i2 < n2 :
8.          if i1 >= n1:
9.              L.append(L2[i2])
10.             i2 = i2 + 1
11.         elif i2 >= n2:
12.             L.append(L1[i1])
13.             i1 = i1 + 1
14.         else:
15.             e1 = L1[i1]
16.             e2 = L2[i2]
17.
18.
19.
20.
21.
22.
23.     return L
```

Dans cette fonction, les entiers `i1` et `i2` représentent respectivement les indices des éléments des listes `L1` et `L2` que l'on souhaite comparer :

- Si aucun des deux indices n'est valide, la boucle `while` est interrompue ;
- Si `i1` n'est plus un indice valide, on va ajouter à `L` les éléments de `L2` à partir de l'indice `i2` ;
- Si `i2` n'est plus un indice valide, on va ajouter à `L` les éléments de `L1` à partir de l'indice `i1` ;
- Sinon, le plus petit élément non encore traité est ajouté à `L` et on décale l'indice correspondant.

Écrire sur la copie les instructions manquantes des lignes 17 à 22 permettant d'insérer dans la liste `L` les éléments des listes `L1` et `L2` par ordre croissant.

EXERCICE 3. : SUJET 2022



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. Recopier et compléter sur la copie 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. Recopier et compléter sur la copie les lignes 6 à 10 de la classe `Labyrinthe`.

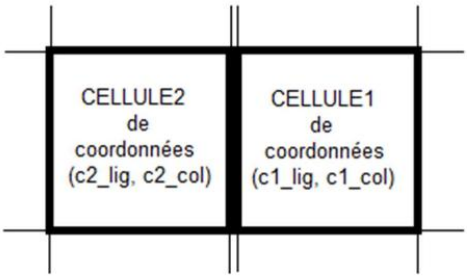
```
1  class Labyrinthe:
2      def __init__(self, hauteur, longueur):
3          self.grille=self.construire_grille(hauteur, longueur)
4      def construire_grille(self, hauteur, longueur):
5          grille = []
6          for i in range(...):
7              ligne = []
8              for j in range(...):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

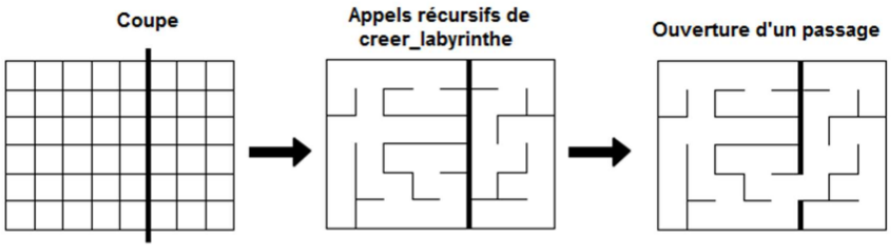
```
13      def creer_passage(self, c1_lig, c1_col, c2_lig, c2_col):
14          cellule1 = self.grille[c1_lig][c1_col]
15          cellule2 = self.grille[c2_lig][c2_col]
16          # cellule2 au Nord de cellule1
17          if c1_lig - c2_lig == 1 and c1_col == c2_col:
18              cellule1.murs['N']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

3. La ligne 18 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 19.

4. Recopier et compléter sur la copie le code Python des lignes 21 à 23 qui permettent le traitement du cas où `cellule2` est à l'Ouest de `cellule1` :

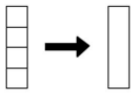


Pour créer un labyrinthe, on utilise la méthode diviser pour régner en appliquant récursivement l'algorithme `creer_labyrinthe` sur des sous-grilles obtenues en coupant la grille en deux puis en reliant les deux sous-labyrinthes en créant un passage entre eux.



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. Recopier et compléter sur la copie les lignes 25 à 30 de la méthode `creer_labyrinthe` traitant le cas de base.

```
24      def creer_labyrinthe(self, ligne, colonne, haut, long):
25          if haut == 1 : # Cas de base
26               for k in range(...):
27                   self.creer_passage(ligne, k, ligne, k+1)
28          elif long == 1: # Cas de base
29               for k in range(...):
30                   self.creer_passage(...)
31          else: # Appels récursifs
32             # 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.