2TSI. TD PYTHON 19. INDICATIONS

Partie A

On rappelle qu'une Pile est LIFO : Last In - First Out (Piles d'assiettes).
Une File est FIFO : First In - First Out (file d'attente à un guichet).
Il faut lire ici les ordres de sortie de gauche à droite. Par ailleurs, l'énoncé n'est pas très clair sur les entrées. Doit-on tout rentrer avant de tout sortir , Vous ferez donc deux hypothèses.
1) Premier cas : On fait rentrer toutes les valeurs dans l'ordre de l'énoncé avant d'en retirer.
2) Second cas : On peut retirer les éléments de la structure entre les opérations d'insertion.

Partie B

1) On rappelle que $L[:j]$ est la sous-liste tirée de L du début à l'indice $j-1$ et $L[i:]$ est la sous-liste tirée de L de l'indice i à la fin.
2) Pour construire *merge* on définit au départ une liste vide *resultat* et en fin de procédure, *resultat* doit être rempli des deux listes $liste1$ et $liste2$ fusionnées et triées.
L'idée est d'adapter les procédures $fusion$ et $TriFusion$ du cours.
On tape en langage PYTHON :
On initialise $i1$ et $i2$ à 0.
Tant que $i1 < len(liste1)$ et $i2 < len(liste2)$ alors soit $liste1[i1] < liste2[i2]$ et alors on ajoute $liste1[i1]$ à *resultat* et on ajoute 1 à $i1$, soit $liste1[i1] >= liste2[i2]$ et alors on ajoute $liste2[i2]$ à *resultat* et on ajoute 1 à $i2$.
Puis après cela, une des deux listes est vide et l'autre reste éventuellement à vider. On écrit une nouvelle boucle conditionnelle : tant que $i1 < len(liste1)$ alors on ajoute $liste1[i1]$ à *resultat* et on ajoute 1 à $i1$.
On fait de même avec la liste $liste2$ si c'est elle qui n'est pas vide.
3) La procédure $merge\_sort(liste)$ est une fonction récursive sur la longueur de la liste.
On commence par taper :
$if\ len(liste) <= 1$:
return liste

Puis dans $liste1$, $liste2$ on rentre $partition(liste)$
Puis on $return\ (merge\_sort(liste1), merge\_sort(liste2))$

Partie C

Étape 1. On commencera par écrire une fonction qui calcule le gap d'une liste.
L'idée est de calculer le minimum *mini* et le maximum *maxi* des éléments d'une liste.
Et on $return\ maxi - mini$
Si vous voulez ne pas utiliser les fonctions prédéfinies pour connaître *mini* et *maxi* alors dans la procédure $gap(liste)$, vous initialisez *mini* et *maxi* à $liste[0]$ puis dans une boucle for indexée par i de 1 à $len(liste)$, vous comparez $liste[i]$ à *mini* et si $liste[i] < mini$ alors on a trouvé le nouveau *mini* et sinon, si $liste[i] > maxi$ on a trouvé le nouveau *maxi*.
Étape 2. On construit la fonction max_gap_matrix d'argument M qui $return\ max\_gap$ c'està-dire le maximum des gaps sur les lignes.
L'idée est de faire comme *gap* mais on indexe sur le numéro de la ligne de la matrice M.
On commence par importer *numpy as np*
Dans max_gap on rentre initialement $gap(M[0])$ puis on fait une boucle for indexée par i de 1 à $np.size(M, 0)$ (qui donne le nombre de lignes de M, d'ailleurs $len(M)$ donne exactement la même chose).
Dans la boucle, on affecte new_gap de $gap(M[i])$ et on compare new_gap et max_gap. Si $new\_gap > max\_gap$ vous avez compris quoi faire.
Version light. On peut utiliser les fonctions *max* et *min* prédéfinies ainsi, $max(ligne)$ donne le maximum sur une ligne.

Partie D

On comprend pourquoi une matrice non carée n'a aucune chance d'être symétrique.
Dans $is\_symetric(A)$ on pose $n = np.size(A, 0)$ et on commence une boucle for indexée par i de 0 jusquà n exclu
On commence par tester si $len(A[i])$ vaut n. Si ce n'est pas le cas, on renvoie $False$
puis dans une nouvelle boucle for (incluse dans la première) indexée par j de $i+1$ à n exclu, on teste si $A[i][j]$ vaut $A[j][i]$ ou non. Si c'est non, on renvoie encore $False$
On rappelle au passage que $a\ != b$ signifie que a est différent de b.
Et après être sorti des boucles, on $return\ True$