I Tri en python

```
# L contient des données ''comparables'' entre elles
L.sort()  # trie la liste L, dans l'ordre croissant.
```

Exercice 1
Compléter la fonction `mediane(L)` qui prend en argument une liste de nombres triée dans l'ordre croissant et retourne la médiane de ces nombres. Si `L` est de longueur paire, on utilisera la moyenne arithmétique des deux éléments centraux.

Exercice 2
Compléter la fonction `recherche_dicho_rec` qui utilise la récursivité pour implémenter l'algorithme de recherche par dichotomie dans une liste triée. Les arguments sont les suivants :

- `L` : une liste triée dans l'ordre croissant
- `e` : l'élément dont on cherche l'indice dans `L`
- `imin` : l'indice à partir duquel on cherche `e`
- `imax` : l'indice jusqu'où on cherche `e` (indice inclus, pas comme dans un `range`)

Le principe est le suivant : pour le premier appel `imin` vaut 0 et `imax` vaut `len(L) - 1` (c'est à dire qu'on cherche `e` dans la liste en entier). A chaque étape, on détermine un ensemble d'indice strictement pus petits dans lequel chercher `e` et on effectue un appel récursif jusqu'à arriver à un cas d'arrêt qui est déjà fourni dans le code.

II Bases de données, approche en python

2.1 Représentation en python

Une table d'une base de données sera représentée en python par une liste de tuples (ou une liste de listes). Chaque tuple représente une ligne de cette table et tous les tuples seront donc de même longueur.
On ne se préoccupera pas ici des noms des champs de chaque table. Les données seront repérées par l'indice de leurs lignes/colonnes.
Par exemple la table `notes`

id	nom	note
1	eleve1	12
2	eleve2	8
3	eleve1	17

sera représentée en python par une liste de longueur 3 (le nombre de lignes présentes dans la table) et dont chaque élément est un tuple (c'est à dire une liste non modifiable) de longueur 3 également (car la table possède 3 colonnes).

```
NOTES = [
(1, 'eleve1', 12),
(2, 'eleve2', 8),
(3, 'eleve1', 17)
]
```

2.2 Rappels sur les jointures

La syntaxe suivante :

```
SELECT (ici des noms de colonnes)
FROM table1
JOIN table2
```

retourne la table qui est le produit cartésien des deux tables `table1` et `table2` (en ne créant que les colonnes voulues, ce qui ne nous préoccupera pas pour l'instant). C'est à dire que pour chaque ligne de `table1`, on la concatène avec chaque ligne de `table2` pour créer une ligne de la table jointure.
Si on considère la table `devoirs`

id	nom	noteId
1	devoir1	1
2	devoir1	2
3	devoir2	3

alors la jointure de ces deux tables est une table composée de 6 colonnes et 9 lignes :

notes.id	notes.nom	notes.note	devoirs.id	devoirs.nom	devoirs.noteId
1	eleve1	12	1	devoir1	1
1	eleve1	12	2	devoir1	2
1	eleve1	12	3	devoir2	3
2	eleve2	8	1	devoir1	1
2	eleve2	8	2	devoir1	2
...	...	...	...	...	...

Exercice 3
Compléter la fonction `produit` qui réalise cette opération. Donner en fonction de l_1 et l_2 (les nombres de lignes respectifs des deux tables) le nombre de lignes de la table produit.

Exercice 4
Le fichier *donnees.py* contient la représentation python d'une base de donnée (concernant des résultats de bac) contenant trois tables :

1. academies
   - id
   - nom

2. departements
   - id (c'est une chaîne de caractères)
   - nom
   - academieId
3. lycees
   - id
   - nom
   - departementId
   - tauxDeReussite
   - ville

```
1  from donnees import ACADEMIES, DEPARTEMENTS, LYCEES
```

Tester la fonction de l'exercice précédent sur les tables `academies` et `departements`. Combien de lignes contiendrait la table produit de ces 3 tables ? Ne pas tester ! Par contre il s'agit de la table à considérer pour faire, par exemple la moyenne des taux de réussites dans l'académie de Rouen.
Évaluer la taille en mémoire de la table produit de `academies` et `departements`. On se servira de la fonction `taille_tuple`

2.3 Jointure avec condition

En pratique, une jointure s'écrit

```
SELECT (ici des noms de colonnes)
FROM table1
JOIN table2
ON table1.c1 = table2.c2
```

où c1 et c2 sont des noms de colonnes dans `table1` et `table2` respectivement. La table construite contient alors uniquement les lignes de la table produit dans lesquelles les valeurs de `table1.c1` et `table1.c2` sont égales. Par convention on ne répète pas la colonne des valeurs égales.

Exercice 5
Compléter la fonction `jointure`. Avec les notations de l'exercice précédent, combien de tests d'égalité entre les valeurs des colonnes devons-nous effectuer ?
Tester cette fonction sur les tables `academies` et `departements`, en imposant l'égalité des colonnes `academies.id` et `departements.academieId`. On doit obtenir autant de lignes qu'il y a de département et 4 colonnes.
Joindre cette table à `lycees` et évaluer la taille de la table obtenue. Comparer à la taille (que vous estimerez) de la table produit (sans créer celle-ci, encore une fois)

Exercice 6
On considère maintenant que les tables sont triées suivant les valeurs des colonnes égales (avant d'effectuer la jointure, le faire dans la console avant si nécessaire : on dispose pour cela de la fonction `tri_colonne` qui trie une table suivant la colonne d'indice donné).
Implémenter un algorithme qui effectue la jointure plus efficacement sous cette hypothèse dans la fonction `jointure_tri`.

2.4 Les autres parties d'une requête

Exercice 7
Compléter la fonction `filtre` qui prend une table, un indice i et une valeur v en entrée et qui retourne la table composée des lignes dont la valeur à l'indice i est v.
Cette fonction nous permet d'implémenter la clause WHERE.

Exercice 8
Même chose pour la fonction `projection` pour la clause SELECT

Exercice 9
Obtenir en python la liste des taux de réussites des lycées de l'académie de "Rouen".
En SQL, on pourrait utiliser

```
SELECT lycees.tauxDeReussite
FROM lycees
JOIN departements on departement.id = lycees.departementId
JOIN academies on academies.id = departements.academieId
WHERE academies.nom = ''Rouen''
```

Calculer ensuite la moyenne, la médiane de ces taux et obtenir les lycées au dessus et les lycées au dessous de la médiane.