Exercice 1 ***8 points***

Partie A : Schéma relationnel

1. `N_Obj` a été choisi comme clé primaire de la table `Gaia` car, d'après l'énoncé, il identifie chaque objet cartographié de manière unique.

2. Schéma relationnel de la table `Typologie` : (`Type` : `String`, `Libelle_Type` : `String`)

Partie B : Instructions SQL

3. **a.**
```
INSERT INTO Gaia VALUES ('8', 4, 'WISEA J085510', 'Naine Brune',
'WISEA J085510', 133.781, -7.244, 439.000, 'WISEA J085510') ;
```
Provoque une erreur car l'attribut `N_Obj` est à `'8'`, qui est une chaine de caractères, alors qu'on attend un entier.

b.
```
INSERT INTO Gaia VALUES (9, 4, 'WISEA J085510', 'Naine Brune',
'WISEA J085510', 133.781, -7.244, 439.000, 'WISEA J085510') ;
```
Provoque une erreur car l'attribut `N_Obj` est à `9`, qui est une valeur déjà existante dans le tableau. Or `N_Obj` est une clé primaire, et donc ne peut pas être la même pour deux enregistrements.

c.
```
INSERT INTO Gaia VALUES (8, 4, 'WISEA J085510', 'Naine Brune',
'WISEA J085510', 133.781, -7.244, 439.000, 'WISEA J085510') ;
```
Ne provoque pas d'erreur.

d.
```
INSERT INTO Gaia VALUES (8, 4, 'WISEA J085510', 'Naine Brune',
'WISEA J085510', '133.781', -7.244, 439.000, 'WISEA J085510') ;
```
Provoque une erreur car l'attribut `Asc_Droite` est à `'133.781'`, qui est une chaine de caractères, alors qu'on attend un nombre réel.

4. Dans la table `Typologie`, l'attribut `'Type'` est une clé primaire. Or cette requête essaie d'insérer un enregistrement ayant un attribut `'Type'` égal à `'BD'`, qui est déjà présent dans la table. Le SGBD refusera donc cette requête.

5. Le résultat de la requête sera :

N_Obj	Parallaxe
2	768 067
11	392 753

6.
```
SELECT Gaia.Nom_Syst, Gaia.Nom_Obj, Typologie.Libelle_Type
FROM Gaia
JOIN Typologie ON Gaia.Type = Typologie.Type
WHERE Gaia.Parallaxe > 400000 AND Typologie.Libelle_Type = 'Etoile de faible masse'
```

Exercice 2 *10 points*

1. Le code proposé est incorrect, car la valeur `lst[i2]` est écrasée dès la première ligne. À la fin de code, les deux éléments `lst[i1]` et `lst[i2]` sont tous les deux égaux à la valeur initiale de `lst[i1]`.

On peut proposer comme modification :

```
def echange(lst, i1, i2):
    temp = lst[i2]
    lst[i2] = lst[i1]
    lst[i1] = temp
```

ou encore

```
def echange(lst, i1, i2):
    lst[i2], lst[i1] = lst[i1], lst[i2]
```

2. Les valeurs qui peuvent être renvoyées par l'appel `randint(0,10)` sont `0`, `1`, `9` et `10`.

3. **a.** La fonction `melange` se termine toujours car l'appel récursif ne se fait que si `ind > 0`. Or, à chaque appel, la variable `ind` est décrémentée de 1. On a donc la certitude qu'au bout d'un certain temps, il n'y aura plus d'appel récursif et donc la fonction se terminera.

b. Pour une liste de longueur n, l'indice maximal de la liste est `n-1`. L'appel initial est donc `melange(lst, n-1)`.

Cet appel va provoquer l'appel à `melange(lst, n-2)`, qui provoquera l'appel à `melange(lst, n- 3)`, et ainsi de suite jusqu'à `melange(lst, 0)`, qui sera le dernier appel.

Le nombre total d'appel (hormis l'appel initial) est donc le nombre d'éléments dans la suite `n-2`, `n-3`, ..., `2`, `1`, `0`. Il y a $n-1$ éléments dans cette suite, et donc il y aura $n-1$ appels récursifs.

c.
```
affichage après appel initial : [0, 1, 2, 3, 4]
affichage après 1er appel récursif et valeur aléatoire égale à 2 : [0, 1, 4, 3, 2]
affichage après 2ème appel récursif et valeur aléatoire égale à 1 : [0, 3, 4, 1, 2]
affichage après 3ème appel récursif et valeur aléatoire égale à 2 : [0, 3, 4, 1, 2]
affichage après 4ème appel récursif et valeur aléatoire égale à 0 : [3, 0, 4, 1, 2]
```

d.
```
def melange_iteratif(lst):
    print(lst)
    ind = len(lst)-1
    while ind > 0:
        j = randint(0, ind)
        echange(lst, ind, j)
        ind -= 1
```

Exercice 3 *12 points*

1. Si les taches doivent être exécutées dans le même ordre que leur ordre d'arrivée dans la structure, la structure de stockage doit avoir le comportement d'une file. En effet, dans une file, l'ordre de passage respecte l'ordre d'arrivée (FIFO), alors que dans une pile, les éléments arrivés en dernier seront traités en premier (LIFO).

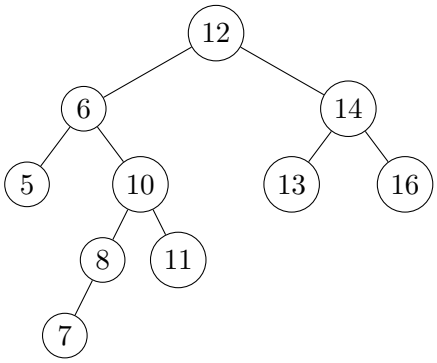
2. Le nombre de tâches restant à effectuer, c'est-à-dire le nombre total de nœuds de l'arbre, s'appelle la taille de l'arbre.

3. **a.** Les attributs de la classe `Noeud` sont `tache`, `indice`, `gauche` et `droite`.

b. La méthode `insere` est récursive car elle fait appel à elle-même dans sa propre définition. Elle se termine car chaque appel récursif se fait sur un sous-arbre, qui finira forcément par être vide. Or si l'appel se fait sur sous-arbre vide, il n'y a plus d'appel récursif (lignes 17 et 18).

c. Comme la ligne 20 lance un appel récursif d'insertion sur le sous-arbre gauche, cela signifie que la valeur à insérer était plus petite que la valeur de la racine. Il faut donc écrire `self.racine.indice > nouveau_noeud.indice`

d. Nouvel arbre :



4.
```
def est_present(self, indice_recherche) :
    """renvoie True si l'indice de priorité indice_recherche (int) passé en paramètre
    ↪  est déjà l'indice d'un noeud de l'arbre, False sinon"""
    if self.racine is None:
        return False
    if self.racine.indice == indice_recherche:
        return True
    if self.racine.indice > indice_recherche:
        return self.racine.gauche.est_present(indice_recherche)
    else:
        return self.racine.droite.est_present(indice_recherche)
```

5. **a.** Parcours infixe de l'arbre binaire de recherche de la FIGURE 1 : `6-8-10-12-13-14`

b. Le parcours infixe donnant l'indice de priorité par ordre croissant, il suffit de prendre la première tache du parcours : c'est celle qui aura l'indice de priorité le plus faible, donc celle à traiter en premier.

6.
```
def tache_prioritaire(self):
    """renvoie la tache du noeud situé le plus à gauche de l'ABR supposé non vide"""
    if self.racine.gauche.est_vide(): #pas de noeud plus à gauche
        return self.racine.tache
    else:
        return self.racine.gauche.tache_prioritaire()
```