25NSIJ1G11

**Exercice 1** (6 points)

*Cet exercice porte sur les graphes et la programmation orientée objet.*

**Partie A**

1. Le code `balise12 = Balise(12, ['vert', 'noir'])` permet d'instancier la balise 12.
2. `balise9.voisines = [balise8, balise9, balise12]` permet de compléter l'implémentation du graphe.
3. Le résultat est `[2, 5, 6]`. La méthode 1 renvoie les numéros des balises voisines de la balie considérée.
4. Le résultat est `['noir', 'vert']`. La méthode 2 permet d'enlever une couleur à une balise, tandis que la méthode 3 permet d'ajouter une couleur à la balise.
5. On complète les lignes 6 à 9 de la fonction `itineraire`.

```
6       while balise.num_balise != balise_fin.num_balise :
7           for b in balise.voisines :
8               if (couleur in b.couleurs_balise) and (b not in chemin):
9                   balise = b
```

*Remarquons que ce code est incorrect dans le cas où il y a une bifurcation, c'est-à-dire lorsque plusieurs balises peuvent compléter un chemin, car alors toutes ces balises seront appendues au chemin, même si seule la dernière servira à continuer le chemin.*

6. Le résultat du parcours en profondeur est 1, 2, 4, 5, 10, 7, 6, 3, 11, 9, 8, 12.

**Partie B**

7. La ligne 33 devient `balise4.voisines = [(balise2, 13), (balise5, 21), (balise6, 15)]`.
8. Le résultat est `balise5`. La fonction `mystere` renvoie la balise la plus proche de la balise passée en paramètre, parmi les balises dont l'attribut `visitee` est `False`.
9. Le parcours glouton sera 1, 2, 4, 6, 11, 9, 12 (de durée totale $5 + 13 + 15 + 8 + 16 + 4 = 61$ minutes).
10. On complète la fonction `itineraire_trail`.

```
1   def itineraire_trail(balise_debut, balise_fin):
2       balise_debut.visitee = True
3       balise = balise_debut
4       chemin= [balise]
5       while balise_fin not in chemin:
6           prochaine = mystere(balise)
7           if prochaine != None:
8               prochaine.visitee = True
9               balise = prochaine
10              chemin.append(balise)
11          else:
12              return None
13      return [b.num_balise for b in chemin]
```

11. **Proposition A** L'algorithme est un algorithme glouton.
12. Un avantage d'un algorithme glouton est qu'il a en général une complexité raisonnable et donne une solution rapidement, par contre un algorithme glouton n'est pas forcément optimal.

**Exercice 2** (6 points)

*Cet exercice porte sur l'algorithmique, la récursivité et les files.*

1. Après l'instruction `v = plateau[1][2]`, la variable `v` vaut `6`.
2. On complète le programme.

```
def plateau_init(n, m, cartes):
    shuffle(cartes)
    plateau = []
    for i in range(n):
        plateau.append([cartes[i+j*n] for j in range(m)])
    return plateau
```

3. On complète la fonction.

```
def carte_voisines(n, m, i, j):
    voisines = []
    for i2 in range(i-1, i+2):
        for j2 in range(j-1, j+2):
            if (i2, j2) != (i,j) and i2 in range(n) and j2 in range(m):
                voisines.append((i2, j2))
    return voisines
```

4. La valeur de `e1` est $9+8+6+7=30$, `e2` n'a pas de valeur car la chaîne n'existe pas, les cartes `(1, 2)` et `(2, 0)` ne sont pas voisines, `e2` n'a pas de valeur car la chaîne n'existe pas, la carte `(1, 4)` n'existe pas.
5. On écrit la fonction demandée.

```
def chaine_evalue(plateau, chaine):
    n, m = len(plateau), len(plateau[0])
    position=chaine[0]
    if position[0] in range(n) and position[1] in range(m):
        valeur = plateau[position[0]][position[1]]
    else:
        return None
    for k in range(1, len(chaine)):
        if chaine[k] in cartes_voisines(n, m, position[0], position[1]):
            position = chaine[k]
            valeur = valeur + plateau[position[0]][position[1]]
        else:
            return None
    return valeur
```

6. Avec un parcours en largeur, on va examiner les chaînes par longueur croissante, ce qui permettra d'arrêter la recherche dès qu'on aura trouvée une solution car on saura que cette solution est optimale.
7. On complète les lignes l3, l8, l10 et l15.

```
3       a_visiter = file_init()

8       while not file_est_vide(a_visiter)

10          if chaine_evalue(plateau, chemin) == cible:

15              if (i, j) not in chemin:
```

8. Pour accumuler toutes les solutions trouvées, on peut utiliser une variable `solutions` de type `list`, qu'on initialise à `[]` et qu'on renvoie ligne 20, et la ligne 11 devient `solution.append(chemin)`.
   *Notons qu'on obtient ainsi toutes les solutions, pas seulement les plus courtes.*

**Exercice 3** (8 points)

*Cet exercice porte sur les types de données construits (listes et dictionnaires), sur les arbres binaires de recherche et sur les bases de données.*

1. La ligne `enregistrement['latitude']` affiche la valeur `38.38825`.
2. La boucle des lignes 4 et 5 extrait la date, la boucle des lignes 6 et 7 extrait l'horaire, l'appel `nettoyage_datetime('2` renvoie le couple `('2024-06-27', '23:36:01')`.

3. Les appels `len(frames)` et `len(frames[1])` renvoient `4` et `5` respectivement.
4. On écrit une fonction pour détecter une éventuelle anomalie d'altitude.

```
def detecter_anomalie(element):
    altitude = element['altitude']
    return altitude < 0 or altitude > 35000
```

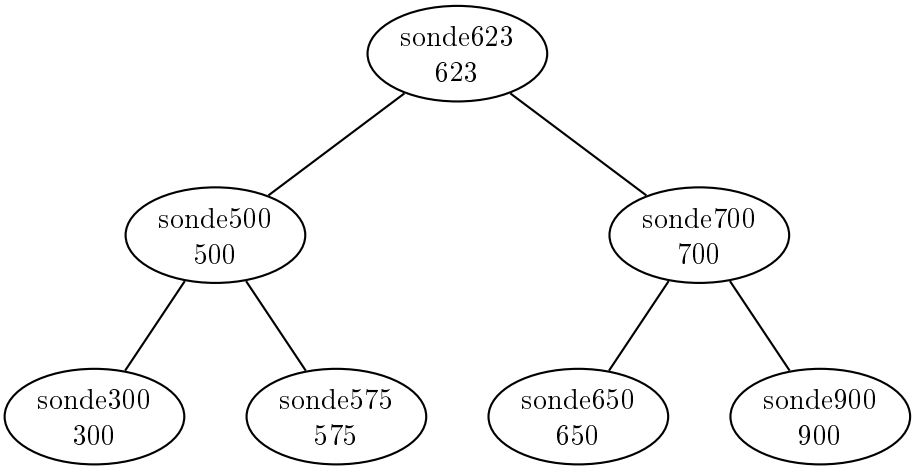
5. On écrit la fonction demandée.

```
def liste_num_serie(frames):
    numeros = []
    for element in frames:
        num = element['num_serie']
        if num not in numeros:
            numeros.append(num)
    return numeros
```

6. On complète la fonction `distance_totale`.

```
def distance_totale(dep):
    total = 0
    for i in range(1, len(dep)):
        total = total + distance_haversine(dep[i-1], dep[i])
    return total
```

7. `sonde623 = Sonde(623, 38.38825, 27.09004, '2024-06-27', sonde500, sonde700)`
8. On complète l'arbre.




9. C'est un parcours infixe qui permet de parcourir l'ABR dans l'ordre croissant des numéros de série.
10. On complète la méthode.

```
    def rechercher(self, numero):
        if self.est_vide():
            return False
        if numero == self.num_serie:
            return self
        elif numero < self.num_serie:
            return self.gauche.rechercher(numero)
        else:
            return self.droit.rechercher(numero)
```

11. La méthode `rechercher` est une méthode récursive car elle s'appelle elle-même.
12. Dans la relation `abonne`, l'attribut `id_abonne` peut servir de clé primaire s'il est unique, c'est-à-dire si chaque ligne a un `id_abonne` distinct.
13. Dans la relation `infos_recuperation`, les clés `num_serie` et `id_abonne` sont des clés étrangères qui référencent respectivement les relations `sonde` et `abonne`.

14. La requête `SELECT nom, prenom FROM abonne WHERE id_abonne>20;` renvoie la table :

| nom | prenom |
|---|---|
| 'Détoile' | 'Diane' |
| 'Girard' | 'Antoine' |

15. On ajoute les données demandées dans la relation `info_recuperation` avec la requête `INSERT INTO info_recuperation VALUES (14,480,24,'2024-07-10',47.230,12.244);`
16. La requête `SELECT num_serie, nom, date_recup FROM info_recuperation JOIN abonne ON info_recuperation.id_abonne = abonne.id_abonne;` convient.