25NSIJ1AN1

Exercice 1 (6 points)

Cet exercice porte sur les arbres binaires, la récursivité et la programmation orientée objet.

1. Le végétal observé est un Sorbier.
2. Les renseignements fournis ne permettent pas de conclure.
3. On écrit le code demandé, en commençant par les feuilles et en terminant par la racine.

```
feuille1=Feuille_resultat([])
feuille2=Feuille_resultat(["Sorbier"])
feuille3=Feuille_resultat(["Robinier","Noyer"])
noeud1=Noeud("Bord denté ?",feuille2,feuille3)
feuille4=Feuille_resultat([])
noeud2=Noeud("Alternées ?",noeud1,feuille4)
arbre_2=Noeud("Simples ?",feuille1,noeud2)
```

On écrit les méthodes demandées.

4.
```
class Noeud:
    ...
    def est_resultat(self):
        return False
```

5.
```
class Feuille_resultat:
    ...
    def est_resultat(self):
        return True
```

6.
```
class Feuille_resultat:
    ...
    def nb_vegetaux(self):
        return len(self.vegetaux)
```

7.
```
class Noeud:
    ...
    def nb_vegetaux(self):
        return self.sioui.nb_vegetaux()+self.sinon.nb_vegetaux()
```

8.
```
class Feuille_resultat:
    ...
    def liste_questions(self):
        return []
```

9.
```
class Noeud:
    ...
    def liste_questions(self):
        return [self.question]+self.sioui.liste_questions()+self.sinon.liste_questions()
```

10. On écrit une fonction `est_bien_renseigne`.

```
def est_bien_renseigne(dico_vegetal,arbre):
    questions=arbre.liste_questions()
    for q in questions:
        if q not in dico_vegetal:
            return False
    return True
```

11. On écrit une fonction `identifier_vegetaux`.

```
def identifier_vegetaux(dico_vegetal,arbre):
    while not arbre.est_resultat():
        if dico_vegetal[arbre.question]:
            arbre=arbre.sioui
        else:
            arbre=arbre.sinon
    return arbre.vegetaux
```

Exercice 2 (6 points)

Cet exercice porte sur la programmation orientée objet, la récursivité et les algorithmes gloutons.

1. On écrit la méthode `passer_transit` de la classe `Colis`.

```
class Colis:
    ...
    def passer_transit(self):
        self.etat='transit'
```

2. On modifie la fonction `ajouter_colis` afin qu'elle ne rajoute pas les colis de plus de 25 kg, avec le répugnant effet de bord consistant à afficher un message d'erreur plutôt que de lever une exception (oui, c'est un jugement de valeur).

```
def ajouter_colis(liste,colis):
    if colis.poids<=25:
        liste.append(colis)
    else:
        print("Dépassement du poids maximal autorisé") # beurk
```

3. `nb_colis=len` est adapté. On peut imaginer que la réponse attendue est plutôt celle présentée ci-dessous.

```
def nb_colis(liste):
    return len(liste)
```

4. On a complété la fonction `poids_total`.

```
def poids_total(liste):
    total=0
    for c in liste:
        total=total+c.poids
    return total
```

5. On écrit une fonction `liste_colis_etat`.

```
def liste_colis_etat(liste,statut):
    res=[]
    for c in liste:
        if c.etat==statut:
            res.append(c)
    return res
```

ou bien

```
def liste_colis_etat(liste,statut):
    return [c for c in liste if c.etat==statut]
```

6. Le tri utilisé ici est un tri sélection, de quadratique dans le pire des cas (et quadratique dans le meilleur des cas : contrairement à d'autres tris, le tri sélection est toujours quadratique, le nombre d'opérations ne dépend que du nombre de données et pas de l'ordre initial des données).
7. On aurait pu utiliser un tri fusion, de complexité quasi-linéaire en $n\log(n)$, basé sur le brillant principe diviser pour régner.

8. On a complété le code de la fonction `chargement_glouton`.

```
def chargement_glouton(liste, rang, capacite):
    if rang == len(liste):
        return []
    elif liste[rang].poids <= capacite:
        return [liste[rang]] + chargement_glouton(liste, rang+1,capacite-liste[rang].poids)
    else:
        return chargement_glouton(liste, rang+1, capacite)
```

9. La fonction est récursive, elle s'appelle elle-même, et pour les appels récursifs Python utilise une pile de taille finie, qui peut déborder si la profondeur de récursion est excessive.
10. On écrit une fonction itérative `chargement_glouton2`.

```
def chargement_glouton2(liste,capacite):
    colis_a_charger=[]
    for c in liste:
        if c.poids<=capacite:
            colis_a_charger.append(c)
            capacite=capacite-c.poids
    return colis_a_charger
```

Exercice 3 (8 points)

Cet exercice porte sur les graphes, les bases de données, les tris, les algorithmes gloutons et la récursivité.

Partie A

1. Le type de l'attribut `annee` de la table `enfant` est `INT`.
2. On pourrait imposer que cet attribut doive appartenir à un intervalle compatible avec le statut d'enfant, entre 0 et 18 ans d'après l'énoncé.
3. Une contrainte de référence qui s'applique à la table `enfant` est que le numéro de téléphone doit être présent dans une ligne de la table `parent`.
4. Le numéro de téléphone `tel` fera une excellente clé primaire pour la table `parent`.
5. La requête `UPDATE parent SET tel=33619782812 WHERE tel=33600782812;` lève une erreur de contrainte d'intégrité car la clé qu'on veut modifier est clé primaire de la table `parent`.
6. On complète la séquence de requêtes.

```
INSERT INTO parent VALUES ('Bauges', 33619782812, 73340);
UPDATE enfant SET num_parent = 33619782812 WHERE num_parent = 33600782812;
DELETE FROM parent WHERE tel = 33600782812;
```

7. La requête `SELECT prenom FROM enfant WHERE annee<2014 ORDER BY annee` fournit le résultat suivant, à partir de l'extrait de la table `enfant` fourni.

prenom
Nakamura
Hawa
Kian
Adrien

8. La requête `SELECT prenom FROM enfant WHERE num_parent=3619861122 ORDER BY prenom;` convient.
9. On utilise ici une jointure des deux tables `enfant` et `parent`.

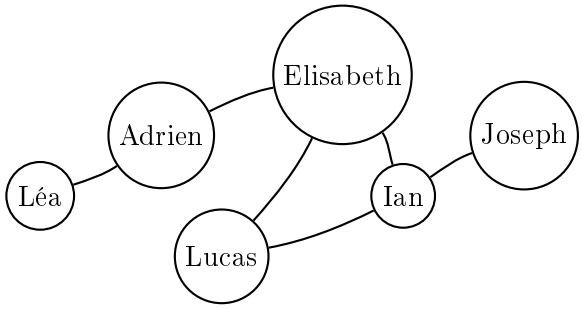
```
SELECT id, prenom FROM enfant
JOIN parent ON parent.tel=enfant.num_parent
WHERE parent.codep=38520;
```

Cette requête permet d'obtenir les identifiants et les les prénoms des enfants dont le parent habite dans une des huit villes qui partagent le code postal 38520, La Garde, Le Bourg d'Oisans, Ornon, Oulles, Saint-Christophe en Oisans, Vénosc, Villard Notre Dame, Villard Reymond.

Partie B

10. La relation « les enfants ne s'entendent pas » est symétrique et peut être modélisée par un graphe non orienté.

11. On a représenté ci-dessous le graphe `g2`.



12. On écrit une fonction `degre`.

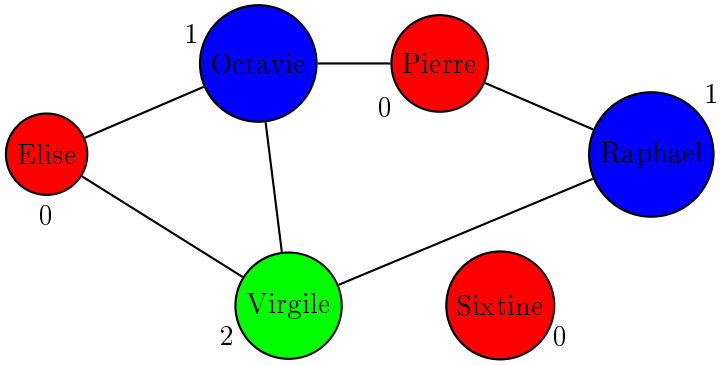
```
def degre(g,s):
    return len(g[s])
```

13. On complète les ligne demandées.

```
7           while j>=0 and degre(g,sommets[j])<degre(g,sommet_courant):
8               sommets[j+1]=sommets[j]
9               j=j-1
10          sommets[j+1]=sommet_courant
```

14. Le tri insertion utilisé est quadratique en le nombre n d'élément triés, dans le pire des cas (par contre il est linéaire sur un tableua déjà trié).

15. On colorie le graphe avec trois couleurs (le triangle Elise Octavie Virgile implique que deux couleurs ne suffisent pas).



De façon équivalente, on peut proposer le coloriage sous la forme de ce dictionnaire :
`{'Elise':0,'Octavie':1,'Virgile':2,'Pierre':0,'Raphael':1,'Sixtine':0}`.

16. On complète la fonction `colorer_graphe`.

```
def colorer_graphe(g,dc):
    for s in dc:
        couleur=plus_petite_couleur_hors_voisins(g,dc,s)
        dc[s]=couleur
```

17. On complète la fonction `welsh_powell`

```
def welsh_powell(g):
    dc={s:-1 for s in g}
    sommets=sommets_tries(g)
    for s in sommets:
        dc[s]=plus_petite_couleur_hors_voisins(g,dc,s)
    return dc
```