Math93.com

BAC NSI - Correction Amérique Nord - 2025 - Sujet 1 - NSI

Thèmes des exercices (sur 20 points)

- Exercice 1 (6 points) : les arbres binaires, la récursivité et la programmation orientée objet..
- Exercice 2 (6 points) : la programmation orientée objet, la récursivité et les algorithmes gloutons. .
- Exercice 3 (8 points) : les graphes, les bases de données, les tris, les algorithmes gloutons et la récursivité..
- Lien vers le sujet

Exercice 1. les arbres binaires, la récursivité et la programmation orientée objet.. 6 points

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

Cet exercice porte sur l'identification de végétaux (tilleul, ficus, ...) à partir de caractéristiques de leurs folia (nom scientifique des feuilles d'un végétal) : simples ou complexes, disposées de façon alternée ou non, etc. Par exemple, un tilleul a des folia simples, disposées de façon alternée mais pas en hélice, en forme de cur et à bord denté. Un ficus a également des folia simples et disposées de façon alternée. Cependant elles sont insérées en hélice et sont de forme ovale. Un robinier a des folia complexes, disposées de façon alternée et non dentées.

Pour identifier un végétal à l'aide des caractéristiques de ses folia, on utilise un arbre binaire appelé arbre de décision. Un exemple de tel arbre de décision est partiellement représenté sur la figure 1 ci-dessous (les parties non représentées de cet arbre sont indiquées par des points de suspension).

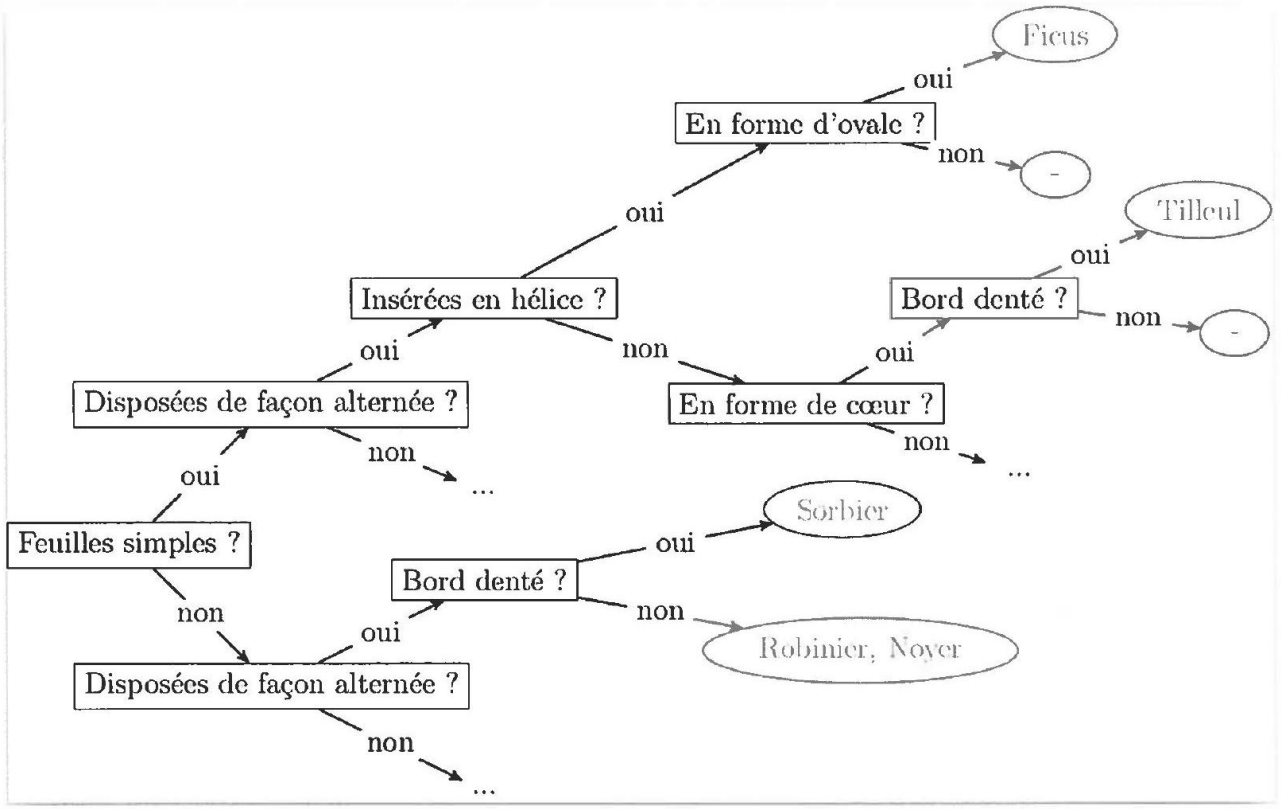

Figure 1. Extrait d'un arbre de décision aidant à reconnaître un végétal à partir des caractéristiques de ses folia.

Les rectangles sont les nuds de l'arbre de décision. Ils correspondent chacun à une question. Les ovales sont les feuilles de l'arbre de décision. lls correspondent chacun à un ensemble de végétaux. Pour chaque question, il est possible de répondre par oui ou par non ce qui permet d'atteindre soit un nouveau nud, c'est-à-dire une nouvelle question, soit une feuille de l'arbre de décision. Cette feuille contient le plus souvent un seul végétal, éventuellement plusieurs si leurs folia ont les mêmes caractéristiques, et éventuellement aucun si aucun végétal connu ne présente ces caractéristiques. Par exemple, robinier et noyer ont tous les deux des folia complexes (non simples), disposées de façon alternée et non dentées : ils sont donc dans la même feuille de l'arbre de décision de la figure 1.

1. On observe un végétal dont les folia sont complexes (non simples), disposées de façon alternée et à bord denté. D'après l'arbre de décision de la figure 1, peut-on identifier ce végétal? Si oui, quel est-il?

Corrigé

On suit les flèches dans l'arbre de décision :

`Feuilles simples ?` → non
`Disposées de façon alternée ?` → oui
`Bord denté ?` → oui

On arrive à la feuille contenant le nom : **Sorbier**.

Conclusion : Oui, le végétal est un **Sorbier**.

2. On observe un végétal dont les folia sont simples, disposées de façon alternée, insérées en hélice et ne sont pas de forme d'ovale. D'après l'arbre de décision de la figure 1, peut-on identifier ce végétal? Si oui, quel est-il?

Corrigé

`Feuilles simples ?` → oui
`Disposées de façon alternée ?` → oui
`Insérées en hélice ?` → oui
`En forme d'ovale ?` → non

On arrive à une feuille vide, représentée par un ovale sans nom.

Conclusion : Non, ce végétal ne peut pas être identifié à partir de l'arbre donné.

L'arbre de décision est représenté en langage Python en utilisant une classe Noeud et une classe Feuille_resultat dont les définitions sont données ci-dessous.

```
class Noeud:
    def __init__(self, question, sioui, sinon):
        self.question = question
        self.sioui = sioui
        self.sinon = sinon
class Feuille_resultat:
    def __init__(self, vegetaux):
        self.vegetaux = vegetaux
```

La classe Noeud a trois attributs :

- un attribut question, qui est une chaîne de caractères représentant une question ;
- un attribut sioui, qui peut être soit un objet de la classe Noeud représentant une autre question, soit un objet de la classe Feuille_resultat ;
- un attribut sinon, qui peut être soit un objet de la classe Noeud représentant une autre question, soit un objet de la classe Feuille_resultat.

La classe Feuille_resultat a un seul attribut, vegetaux, qui est une liste (éventuellement vide) de chaînes de caractères, dans laquelle chaque chaîne est le nom d'un végétal.
Par exemple, pour l'arbre de décision de la figure 1, pour le Noeud dont la question est 'En forme d'ovale ?', l'attribut sioui est un objet de la classe Feuille_resultat dont l'attribut vegetaux est la liste ['Ficus'] alors que l'attribut sinon de ce nud est un objet de la classe Feuille_resultat dont l'attribut vegetaux est la liste vide.

3. Écrire en langage Python le code permettant de construire l'arbre de décision de la figure 2 ci-dessous et de l'affecter à une variable nommée arbre_2.

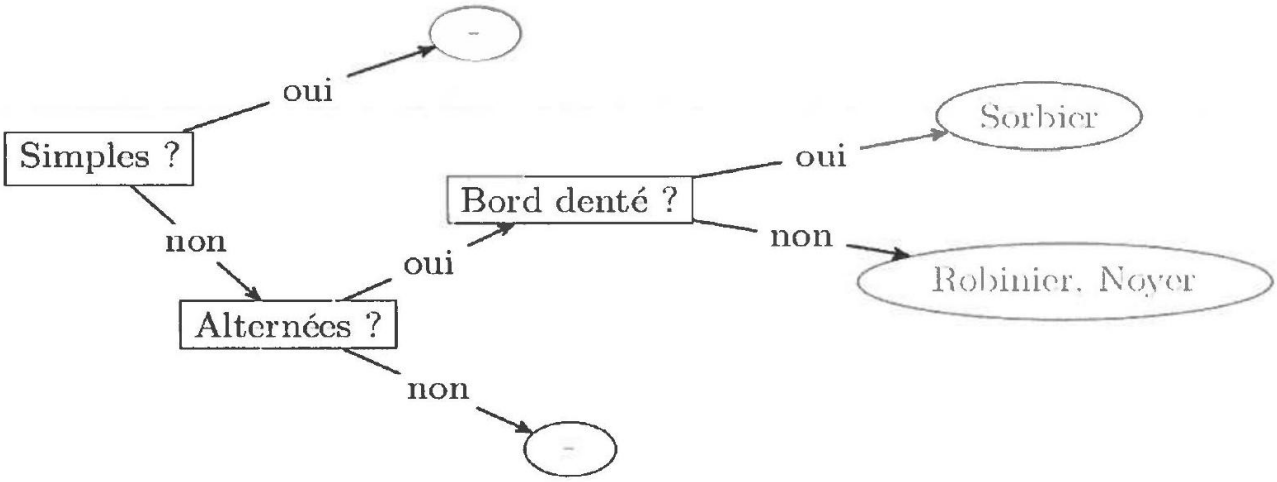

Figure 2. Arbre de décision 2.

Corrigé

```
sorbier = Feuille_resultat(["Sorbier"])
robinier_noyer = Feuille_resultat(["Robinier", "Noyer"])
feuille_vide1 = Feuille_resultat([])
feuille_vide2 = Feuille_resultat([])
n_bord = Noeud("Bord denté ?", sorbier, robinier_noyer)
n_alt = Noeud("Alternées ?", n_bord, feuille_vide1)
arbre_2 = Noeud("Simples ?", feuille_vide2, n_alt)
```

On souhaite écrire une méthode est_resultat pour chacune des classes Noeud et Feuille_resultat. Un objet de la classe Feuille_resultat est un résultat, la méthode doit renvoyer True. Un objet de la classe Noeud n'est pas un résultat, la méthode doit renvoyer False.

4. Écrire le code de la méthode est_resultat pour la classe Noeud.

Corrigé

Un objet de la classe `Noeud` n'est pas un résultat : la méthode renvoie `False`.

```
def est_resultat(self):
    return False
```

5. Écrire le code de la méthode est_resultat pour la classe Feuille_resultat.

Corrigé

Un objet de la classe `Feuille_resultat` est un résultat : la méthode renvoie `True`.

```
def est_resultat(self):
    return True
```

6. Écrire le code de la méthode nb_vegetaux pour la classe Feuille_resultat.

Corrigé

Une feuille contient une liste de végétaux : on retourne sa longueur.

```
def nb_vegetaux(self):
    return len(self.vegetaux)
```

7. Écrire le code de la méthode nb_vegetaux pour la classe Noeud.

Corrigé

Un `Noeud` contient deux sous-arbres : on retourne la somme des végétaux des deux sous-arbres.

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

8. Écrire le code de la méthode liste_questions pour la classe Feuille_resultat.

Corrigé

Une feuille ne contient aucune question : on retourne une liste vide.

```
def liste_questions(self):
    return []
```

9. Écrire le code de la méthode liste_questions pour la classe Noeud.

Corrigé

Un noeud contient une question et deux sous-arbres. On retourne une liste composée de sa propre question suivie de toutes les questions de ses enfants.

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

10. Écrire une fonction est_bien_renseigne...

Corrigé

On parcourt toutes les questions présentes dans l'arbre et on vérifie qu'elles sont bien dans le dictionnaire.

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

11. Écrire une fonction identifier_vegetaux...

Corrigé

On descend dans l'arbre en suivant les réponses du dictionnaire. Lorsqu'on atteint une feuille, on retourne la liste des végétaux.

```
def identifier_vegetaux(dico_vegetal, arbre):
    if arbre.est_resultat():
        return arbre.vegetaux
    if dico_vegetal[arbre.question]:
        return identifier_vegetaux(dico_vegetal, arbre.sioui)
    else:
        return identifier_vegetaux(dico_vegetal, arbre.sinon)
```

Exercice 2. la programmation orientée objet, la récursivité et les algorithmes gloutons. 6 points

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

Une entreprise souhaite gérer les colis qu'elle expédie à l'aide d'une application informatique. On sait que chaque colis a un identifiant unique, un poids, une adresse de livraison et un état. Pour chacun d'entre eux, trois états sont possibles : "préparé", "transit" ou "livré".
Pour cela, on a créé une classe Colis avec les attributs suivants :

- id : un identifiant unique (de type str) ;
- poids : le poids du colis en kilogrammes (de type float) ;
- adresse : l'adresse de destination (de type str) ;
- etat : l'état du colis (de type str parmi 'préparé', 'transit', 'livré').

Lorsque l'on crée une instance de la classe Colis, l'attribut etat est initialisé à 'préparé' tandis que les valeurs des autres attributs sont passées en paramètres.
Voici le début du code Python de la classe Colis :

```
class Colis:
    def __init__(self, id, poids, adresse):
        self.id = id
        self.poids = poids
        self.adresse = adresse
        self.etat = 'préparé'
```

On crée, par exemple, les deux colis suivants :

```
colisA = Colis('AC12', 5.0, '20 rue de la paix 57000 Metz')
colisB = Colis('AF34', 10.25, '32 rue du centre 57000 Metz')
```

1. Écrire la méthode passer_transit de la classe Colis qui permet de mettre l'état du colis à la valeur 'transit'.

Corrigé

La méthode met à jour l'attribut `etat` à la valeur 'transit'.

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

2. Modifier la fonction ajouter_colis pour limiter le poids à 25 kg.

Corrigé

On ajoute une condition pour refuser les colis trop lourds.

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

3. Écrire une fonction nb_colis qui retourne le nombre de colis.

Corrigé

Il suffit de retourner la longueur de la liste.

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

4. Compléter la fonction poids_total.

Corrigé

On initialise `total` à 0 et on ajoute chaque poids.

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

5. Fonction liste_colis_etat

Corrigé

On sélectionne les colis qui ont l'état correspondant.

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

#ou encore

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

6. Quel tri est utilisé dans tri_decroissant?

Corrigé

C'est un tri par sélection (selection sort), dont le coût est quadratique : $O(n^2)$ dans le pire des cas.

7. Autre algorithme de tri possible.

Corrigé

On aurait pu utiliser le tri fusion (merge sort) avec un coût $O(n \log n)$ dans le pire des cas, ou le tri par insertion (coût quadratique), ou tout autre tri.

8. Compléter la fonction récursive chargement_glouton.

Corrigé

On traite un colis si son poids est admissible, sinon on passe au suivant.

```
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. Pourquoi une erreur de type RecursionError peut apparaître ?

Corrigé

Si la liste de colis est très longue, la récursion appelle trop de fois la fonction : cela dépasse la profondeur maximale autorisée par Python (environ 1000 appels).

10. Fonction chargement_glouton2 (version itérative)

Corrigé

On parcourt la liste triée, on ajoute un colis si la capacité restante le permet.

```
def chargement_glouton2(liste, capacite):
    charge = 0
    colis_a_charger = []
    for c in liste:
        if charge + c.poids <= capacite:
            colis_a_charger.append(c)
            charge += c.poids
    return colis_a_charger
```

Exercice 3. les graphes, les bases de données, les tris, les algorithmes gloutons et la récursivité.. 8 points

Cet exercice porte sur les graphes, les bases de données, les tris, les algorithmes gloutons et la récursivité.
Une association s'occupe d'enfants de 0 à 18 ans. Elle souhaite pouvoir former des groupes d'enfants qui s'entendent durant les activités proposées.

Partie A Base de données

Dans cette partie, on pourra utiliser les clauses du langage SQL pour :

- construire des requêtes d'interrogation à l'aide de SELECT, FROM, WHERE (avec les opérateurs logiques AND , OR), JOIN . . . ON ;
- construire des requêtes d'insertion et de mise à jour à l'aide de UPDATE, INSERT, DELETE ;
- affiner les recherches à l'aide de DISTINCT, ORDER BY.

On considère une base de données composée des 3 tables suivantes.

- Table parent :
  - nom est le nom de famille du parent ;
  - tel est le numéro de téléphone du parent ;
  - codep est le code postal de la ville où réside le parent.
- Table enfant
  - id est l'identifiant de l'enfant pour l'association ;
  - prenom est le prénom de l'enfant ;
  - num_parent est le téléphone du parent référent (par soucis de simplicité, on suppose qu'un enfant n'est référencé que par un seul parent) ;
  - annee est l'année de naissance de l'enfant.
- Table mesentente :
  - enfant1 est l'identifiant d'un premier enfant ;
  - enfant2 est l'identifiant d'un second enfant.

Ainsi, on considère que deux enfants qui se trouvent sur la même ligne dans la table mesentente ne peuvent pas effectuer de sortie ensemble.
Le schéma relationnel de la BDD est donné en figure 1, avec la convention que les attributs formant une clef primaire sont soulignés tandis que ceux d'une clef étrangère sont précédés d'un croisillon (symbole #) avec une flèche vers l'attribut référencé.

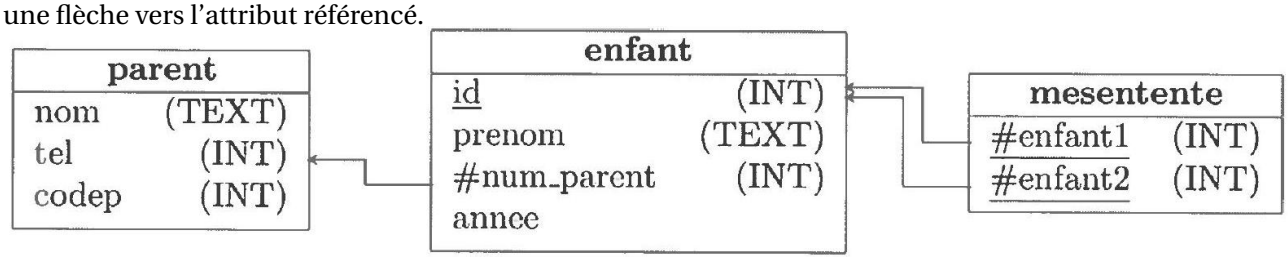

Figure 1. Schéma relationnel de la BDD
On considère la table enfant suivante :

enfant			
id	prenom	num_parent	annee
2	'Hawa'	33619911212	2012
3	'Adrien'	33619861232	2013
6	'Kian'	33619834521	2012
8	'Gabin'	33619847852	2014
12	'Nakamura'	33619732453	2009
14	'Maya'	33600782153	2017
17	'Olivier'	33619868564	2017
21	'Tess'	33619835876	2016
23	'Rachelle'	33600785482	2023

1. Donner le type pour l'attribut annee de la table enfant.

Corrigé

Type : INTEGER, car il s'agit d'une année de naissance entière.

2. Expliquer quelle contrainte de domaine supplémentaire serait pertinente pour cet attribut annee.

Corrigé

Pour être cohérent, on pourrait contraindre ses valeurs de 2007 à 2025 qui correspondrait aux dates de naissances denfants (≤ 18).

3. Donner un exemple d'attribut de la table enfant qui suit une contrainte de référence.

Corrigé

`num_parent` est une clé étrangère qui référence `tel` dans la table `parent`.

4. En expliquant ce choix, proposer une clef primaire pour la table parent.

Corrigé

Le numéro de téléphone portable est un identifiant naturel unique pour chaque parent.

Lattribut tel permet didentifier de façon unique un parent et peut donc être une clef primaire. (Un nom nest pas forcément unique car il y a des homonymes, et le code postal nest pas unique à une personne non plus).

Suite à une mauvaise saisie, le véritable téléphone d'un parent (33619782812) a été transformé en 33600782812. On souhaite corriger cette anomalie avec la requête suivante, mais elle lève une erreur.

```
UPDATE parent SET tel = 33619782812 WHERE tel = 33600782812;
```

5. Expliquer pourquoi la requête proposée lève une erreur.

Corrigé

Il y a violation de contrainte d'intégrité référentielle. La requête SQL échoue car la colonne `tel` est une clé primaire référencée par `num_parent`, donc ne peut pas être modifiée directement.

6. Recopier et compléter alors cette suite de commandes qui permet de changer le numéro de téléphone d'un parent du parent de nom 'Bauges' habitant au code postal 73340 et ayant pour téléphone erroné 33600782812 au lieu de son véritable téléphone 33619782812 :

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

Corrigé

On insère le bon numéro, met à jour les enfants, puis supprime l'ancien.

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

7. En considérant la table enfant fournie, donner le résultat de cette requête SQL.

```
SELECT prenom
FROM enfant
WHERE annee < 2014
ORDER BY annee;
```

Corrigé

Enfants nés avant 2014, triés par année : `'Nakamura', 'Hawa', 'Kian', 'Adrien'`

8. Proposer une requête qui renvoie les prénoms, par ordre alphabétique, des enfants inscrits pour le parent dont le numéro de téléphone est 3619861122.

```
SELECT prenom
FROM enfant
WHERE num_parent = 3619861122
ORDER BY prenom;
```

9. Proposer une requête qui liste les identifiants et prénoms des enfants dont le parent habite dans la ville de code postal 38520.

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

Partie B Graphes et algorithmes

Afin de faciliter en amont les préparations des sorties, on souhaite construire le graphe non orienté des mésententes entre les enfants de l'association. Le graphe est représenté par un dictionnaire dont les clés sont les sommets du graphe, et qui associe à chaque sommet le tableau (type List en Python)

de ses voisins.

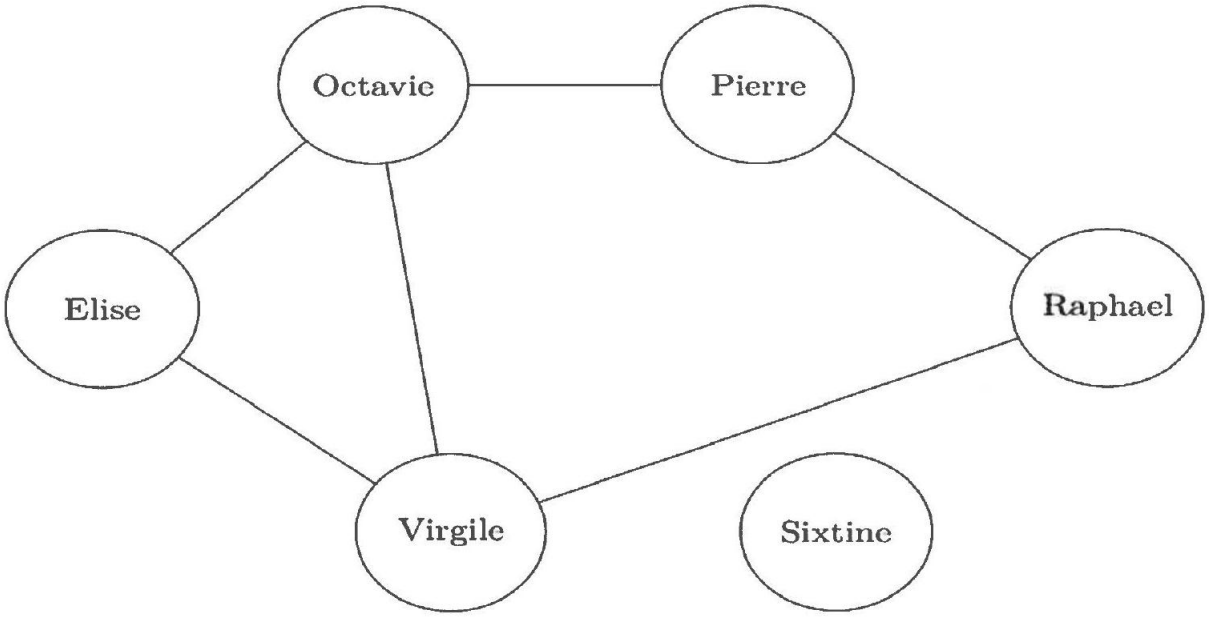

Figure 2. Graphe g1

Par exemple, le graphe g1 de la figure 2 est représenté par le dictionnaire suivant :

```
g1 = {'Elise': ['Octavie', 'Virgile'],
     'Octavie': ['Elise', 'Pierre', 'Virgile'],
     'Pierre': ['Octavie', 'Raphael'],
     'Raphael': ['Pierre', 'Virgile'],
     'Sixtine': [],
     'Virgile': ['Elise', 'Octavie', 'Raphael']
     }
```

10. Expliquer pourquoi la situation décrite ne nécessite qu'un graphe non orienté.

Corrigé

Les mésententes sont réciproques : si A ne peut pas être avec B, alors B ne peut pas être avec A. Il s'agit donc d'un graphe non orienté.

11. Dessiner le graphe g2 défini ci-dessous.

```
g2 = {'Adrien': ['Elisabeth', 'Lea'],
         'Elisabeth': ['Adrien', 'Ian', 'Luca'],
         'Ian': ['Elisabeth', 'Joseph', 'Luca'],
         'Joseph': ['Ian'],
         'Lea': ['Adrien'],
         'Luca': ['Elisabeth', 'Ian']
     }
```

12. Écrire une fonction degre, qui prend en arguments un dictionnaire g représentant un graphe et une chaîne de caractères s représentant un sommet du graphe, et qui renvoie le degré du sommet s dans g. On rappelle que le degré d'un sommet est le nombre d'arêtes issues de ce sommet.

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

13. Recopier et compléter les lignes 7 à 10 de la fonction sommets_tries, qui prend en paramètre un dictionnaire g représentant un graphe, et qui renvoie la liste des sommets du graphe triés dans l'ordre décroissant de leur degré.

```
1  def sommets_tries(g):
2      sommets = [sommet for sommet in g]
3      n = len(sommets)
4      for i in range(1, n):
5          sommet_courant = sommets[i]
6          j = i - 1
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
11     return sommets
```

14. Préciser le tri utilisé dans la question précédente, ainsi que son coût d'exécution en temps dans le pire des cas selon le nombre n de sommets (constant, logarithmique soit en $\log_2(n)$, linéaire soit en n, quasi-linéaire soit en $n\log_2(n)$, quadratique soit en n^2, cubique soit en n^3, exponentiel soit en 2^n,...). On fait l'hypothèse pour cette question que la fonction degre est de coût constant.

Corrigé

Le tri utilisé est le tri par insertion. Dans le pire des cas, son coût est quadratique : $O(n^2)$.

Pour faire des groupes de personnes qui peuvent s'entendre, une méthode consiste à colorer le graphe, c'est-à-dire attribuer une couleur à chacun de ses sommets, en prenant garde qu'aucune arête ne relie deux sommets de même couleur. Ainsi les
sommets d'une même couleur forment un groupe de personnes qui peuvent s'entendre. Par la suite, les couleurs sont représentées par des nombres entiers positifs, et -1 représente l'absence de couleur.

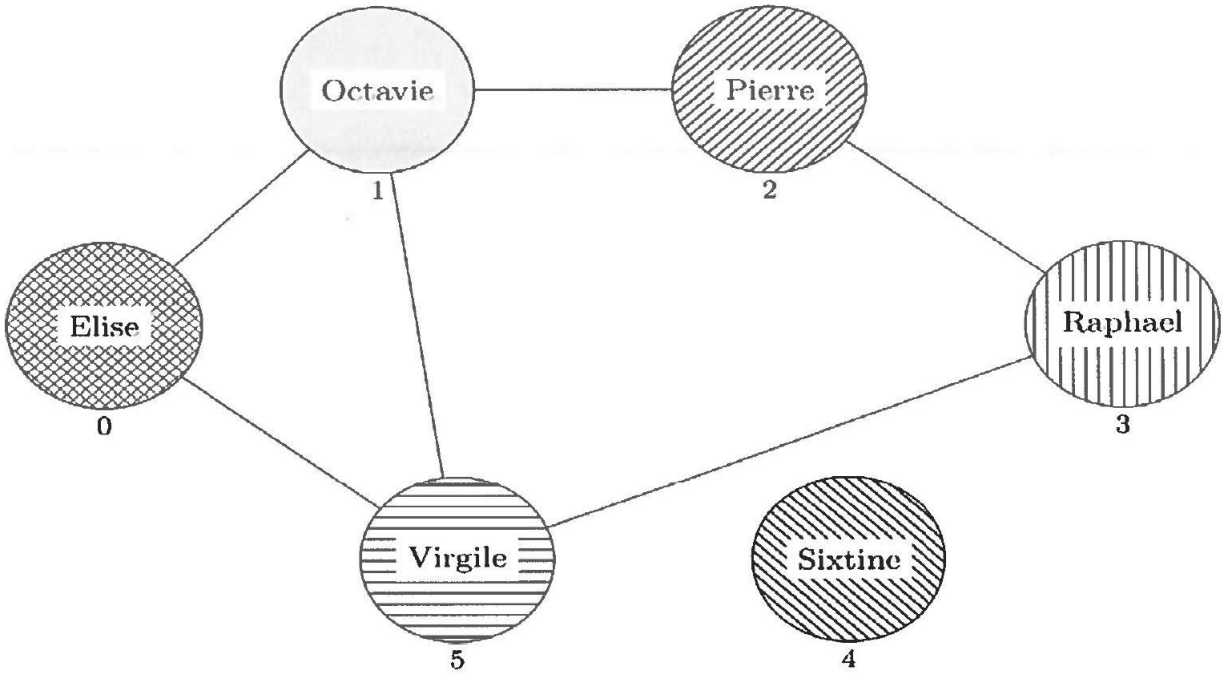

Figure 3. Graphe g1 coloré

En notant les couleurs par différents nombres précisés sous les sommets, le graphe g1 ci-dessus est associé au dictionnaire des couleurs dc1 suivant : dc1 = {'Elise' : 0, 'Octavie' : 1, 'Pierre' : 2, 'Raphael' : 3, 'Sixtine' : 4, 'Virgile' : 5}

On peut utiliser moins de couleurs dans cet exemple.

15. Recopier et colorer le graphe g1 en n'utilisant que trois couleurs (0,1 et 2).

. Corrigé

Une méthode simple pour colorer le graphe consiste à parcourir les sommets et numéroter (colorer) chaque sommet s par le plus petit numéro non utilisé par ses voisins.
On dispose de la fonction plus_petite_couleur_hors_voisins, qui prend en paramètres

- un dictionnaire g représentant un graphe,
- un dictionnaire de couleurs dc dont les clés sont des sommets de g,
- une chaîne de caractères s correspondant à un sommet de g, et qui renvoie le plus petit numéro non utilisé dans dc par les voisins du sommet s.

On remarque que dans l'implémentation utilisée, les couleurs du graphe sont nécessairement numérotées entre 0 et $n-1$ (n étant le nombre de sommets).

```
def couleurs_voisins(g, dc, s):
    return [dc[v] for v in g[s]]
def plus_petite_couleur_hors_voisins(g, dc, s):
    couleur = 0
    n = len(g)
    cvoisins = couleurs_voisins(g, dc, s)
    while couleur < n:
        if couleur not in cvoisins:
            return couleur
        couleur = couleur + 1
    return couleur # au cas où len(dc) = 0
```

16. Recopier et compléter la procédure qui permet de colorer le graphe en modifiant le dictionnaire dc pour qu'il associe finalement à chaque sommet de g sa couleur.

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

```
def colorer_graphe(g, dc):
    # Pré-condition : les clés de dc sont les sommets
    # de g, et les valeurs de dc sont toutes à -1
    for s in dc:
        couleur = ...
        ... = couleur
```

On remarque que la procédure précédente colore les sommets du graphe dans l'ordre donné par les clefs du dictionnaire dc. L'algorithme de Welsh-Powell consiste à colorer le graphe dans l'ordre des sommets par degré décroissant.

17. Recopier et compléter le code de la fonction welsh_powell donné ci-après. Cette fonction prend en paramètre un dictionnaire g correspondant à un graphe, et le colore selon l'algorithme de Welsh-Powell (c'est-à-dire qu'elle renvoie le dictionnaire des couleurs associé).
On pourra s'inspirer de la fonction colorer_graphe donnée ci-dessus et utiliser la fonction sommets_tries.

```
def welsh_powell(g):
    # initialisation à -1 pour tous les sommets dans le
        dictionnaire dc
    dc = ... # possiblement plusieurs lignes
    # coloration en suivant l'approche de Welsh-Powell
    for ...
        ...
    return dc
```

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

↩ Fin du devoir ↪