Bac Général Maths + NSI (Mathématiques et Numérique et sciences informatiques)

SUJET D'EXAMEN (SESSION OFFICIELLE)

Numérique et sciences informatiques - Session 2025

Sujet de l'examen national tel que publié pour la session indiquée. Source : sujets nationaux publiés par le ministère chargé de l'examen et les services académiques. Document mis à disposition par FormaV pour la révision.

Toutes les annales et leurs corrigés : https://bac-maths-nsi.fr/annales-sujets-et-corriges

BACCALAURÉAT GÉNÉRAL

ÉPREUVE D'ENSEIGNEMENT DE SPÉCIALITÉ

SESSION 2025

NUMÉRIQUE ET SCIENCES INFORMATIQUES

JOUR 1

Durée de l'épreuve : **3 heures 30**

L'usage de la calculatrice n'est pas autorisé.

Dès que ce sujet vous est remis, assurez-vous qu'il est complet.

Ce sujet comporte 15 pages numérotées de 1/15 à 15/15.

Le sujet est composé de trois exercices indépendants.

Le candidat traite les trois exercices.

Exercice 1 (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 cœur 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).

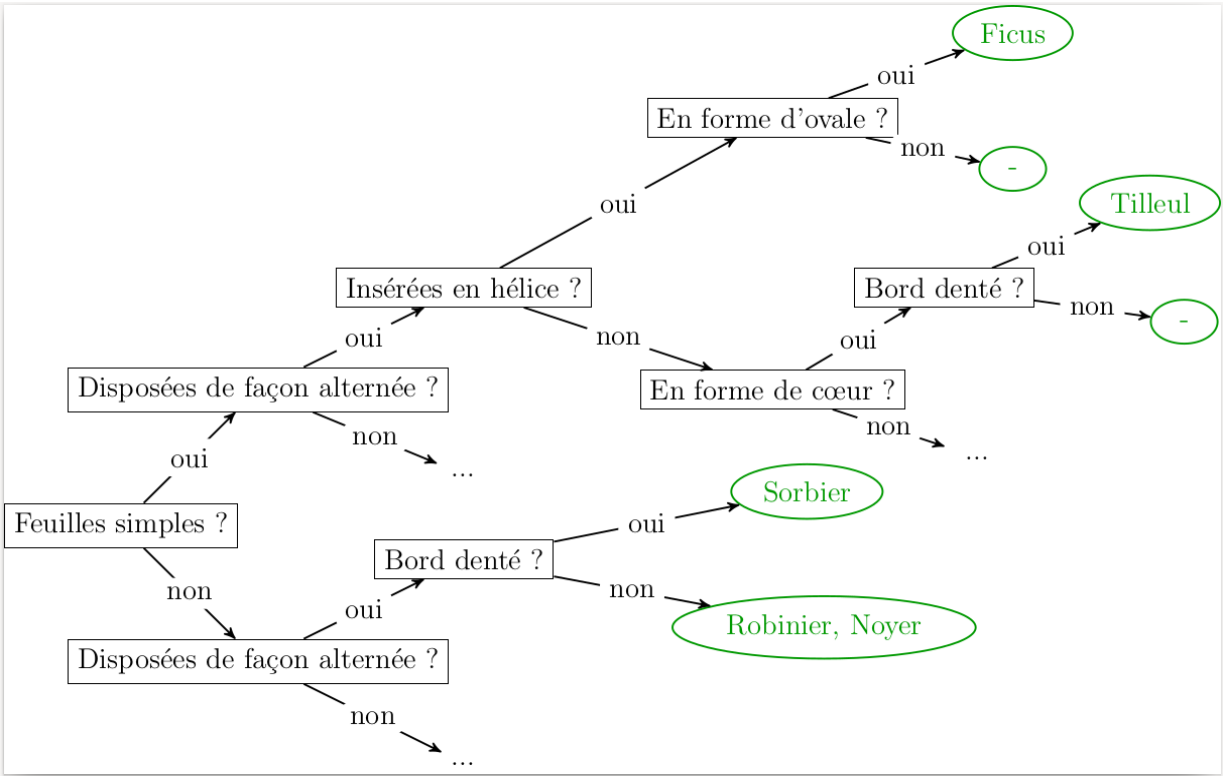


Figure 1. Extrait d'un arbre de décision aidant à reconnaître un végétal à partir des caractéristiques de ses *folia*.

Les rectangles sont les **nœuds** de l'arbre de décision. Ils correspondent chacun à une question. Les ovales sont les **feuilles** de l'arbre de décision. Ils 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 nœud, 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 ?

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 ?

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 nœud 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`.

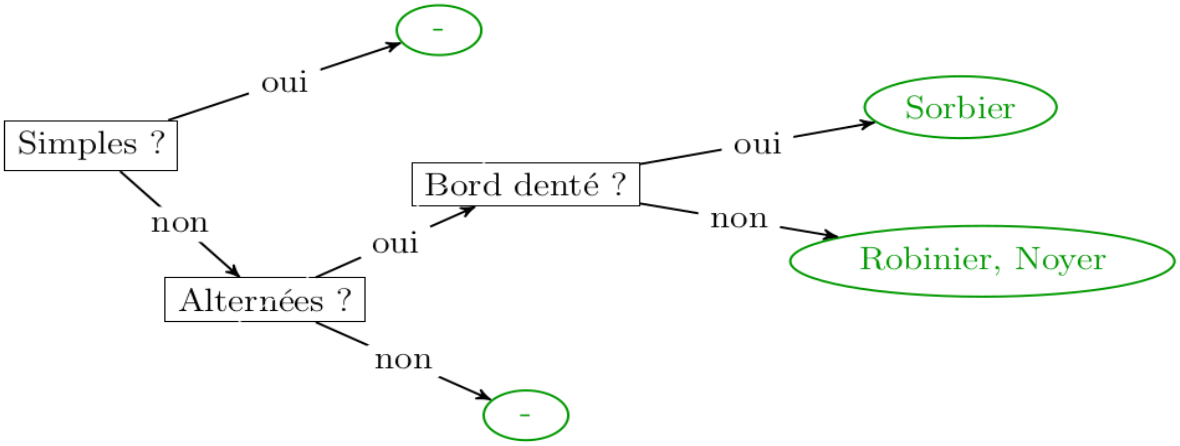


Figure 2. Arbre de décision 2.

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`.

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

On souhaite connaître le nombre de végétaux identifiables par un arbre de décision.

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

7. Écrire le code de la méthode `nb_vegetaux` pour la classe `Noeud`, qui prend en compte tous les végétaux identifiables à partir de ce nœud.

On souhaite enfin écrire une méthode `liste_questions` pour chacune des classes `Noeud` et `Feuille_resultat` afin d'obtenir la liste des questions présentes dans un arbre de décision. L'ordre des éléments dans cette liste n'a pas d'importance, de plus elle peut contenir des doublons. On remarque que :

- si `f` est un objet de la classe `Feuille_resultat`, alors `f.liste_questions()` est la liste vide ;

- le résultat de l'appel `arbre_2.liste_questions()` est la liste `['Simples ?', 'Alternées? ', 'Bord denté ?']` (ou une liste avec les mêmes éléments mais dans un ordre différent).

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

9. Écrire le code de la méthode `liste_questions` pour la classe `Noeud`, qui prend en compte toutes les questions accessibles à partir de ce nœud. On rappelle que l'opérateur + en Python permet de concaténer des listes, par exemple la valeur de l'expression `[1,2]+[3,4,5]` est la liste `[1,2,3,4,5]`.

Pour représenter les caractéristiques des *folia* d'un végétal, on utilise un dictionnaire. Les clés du dictionnaire sont les questions de l'arbre de décision et les valeurs sont `True` ou `False` selon la réponse.

Par exemple, le dictionnaire décrivant les *folia* du sorbier pour l'arbre de décision de la figure 2 est le dictionnaire `folia_sorbier` défini ci-dessous.

```
1 folia_sorbier = {
2   'Simples ?': False,
3   'Alternées ?': True,
4   'Bord denté ?': True
5 }
```

En revanche le dictionnaire `folia_tilleul` ci-dessous qui décrit (partiellement) les *folia* du tilleul n'est pas adapté pour l'arbre de décision de la figure 1 car des données sont manquantes. Par exemple, la question `'Feuilles simples ?'` n'est pas une clé de ce dictionnaire alors que c'est une question présente dans l'arbre.

```
1 folia_tilleul = {
2   'En forme d'ovale ?': False,
3   'Disposées de façon alternée ?': True,
4   'Bord denté ?': True
5 }
```

On cherche à éviter ce genre de cas, afin de ne pas d'utiliser un arbre de décision pour classifier un végétal à partir d'un dictionnaire qui n'est pas assez renseigné.

10. Écrire une fonction `est_bien_renseigne` qui prend en paramètres :

    – un dictionnaire `dico_vegetal` qui donne les caractéristiques des *folia* d'un végétal ,

    – un arbre de décision représenté par un objet `arbre` de la classe `Feuille_resultat` ou de la classe `Noeud`,

et qui renvoie `True` si toutes les questions présentes dans `arbre` sont des clés de `dico_vegetal`.

11. Écrire une fonction `identifier_vegetaux` qui prend en paramètres :

    – un dictionnaire `dico_vegetal` qui donne les caractéristiques des *folia* d'un végétal,

    – un arbre de décision représenté par un objet `arbre` de la classe `Feuille_resultat` ou de la classe `Noeud`,

    et qui renvoie la liste, éventuellement vide, des noms des végétaux dont les *folia* correspondent aux caractéristiques du dictionnaire.

    Par exemple l'appel `identifier_vegetaux(arbre_2,folia_sorbier)` devra renvoyer la liste `['Sorbier']`.

    On suppose que toutes les questions de l'arbre de décision `arbre` apparaissent comme des clés dans le dictionnaire `dico_vegetal`.

Exercice 2 (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` :

```
1 class Colis:
2     def __init__(self, id, poids, adresse):
3         self.id = id
4         self.poids = poids
5         self.adresse = adresse
6         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'`.

On dispose de la fonction `ajouter_colis` suivante :

```
1 def ajouter_colis(liste, colis):
2     # ajoute le colis à la fin de la liste
3     liste.append(colis)
```

Par exemple, après l'exécution des trois instructions suivantes, on a ajouté les deux colis crées précédemment à la liste `liste_colis` :

```
1 liste_colis = []
2 ajouter_colis(liste_colis, colisA)
3 ajouter_colis(liste_colis, colisB)
```

2. Dans cette question uniquement, on considère que l'acheminement des colis de plus de 25 kg est refusé par le transporteur.

   Recopier et modifier le code de la fonction `ajouter_colis` afin qu'elle ajoute le colis à la liste si son poids est inférieur ou égal à 25 kg et qu'elle affiche le message "Dépassement du poids maximal autorisé" sinon.

3. Écrire une fonction `nb_colis` qui prend en paramètre une liste d'objets de la classe `Colis` et qui renvoie le nombre de colis présents dans cette liste.

4. Recopier et compléter les lignes 2 et 4 du code ci-après de la fonction `poids_total` qui prend en paramètre une liste d'objets de la classe `Colis` et qui renvoie le poids total de l'ensemble des colis de cette liste.

```
1 def poids_total(liste):
2     total = ...
3     for c in liste :
4         total = ...
5     return total
```

5. Écrire une fonction `liste_colis_etat` qui prend en paramètres une liste d'objets de la classe `Colis` et une chaîne de caractères `statut` (parmi `'préparé'`, `'transit'` ou `'livré'`) et qui renvoie une nouvelle liste contenant l'ensemble des colis de cette liste dont l'état est le même que `statut`.

L'entreprise tente d'optimiser, à l'aide d'un algorithme glouton, le chargement des colis dans un camion ayant une capacité exprimée en kilogrammes, sans tenir compte de la contrainte de volume. La procédure gloutonne adoptée est la suivante : on charge les colis dans le camion en les choisissant par ordre décroissant de leur poids, sans dépasser la capacité du camion.

Pour cela, il est nécessaire de travailler sur une liste de colis triés par ordre de poids décroissants. La fonction `tri_decroissant` permet de réaliser ce tri.

```
 1 def tri_decroissant(liste):
 2     n = len(liste)
 3     for i in range(n - 1):
 4         min_pos = i
 5         for j in range(i + 1, n):
 6            if liste[j].poids > liste[min_pos].poids:
 7                 min_pos = j
 8         # Échanger les éléments
 9         temp = liste[i]
10         liste[i] = liste[min_pos]
11         liste[min_pos] = temp
12     return liste
```

6. Donner le nom du tri utilisé dans la fonction `tri_decroissant` ainsi que son coût dans le pire des cas.

7. Citer un autre algorithme de tri qui aurait pu être utilisé, ainsi que son coût dans le pire des cas.

Le code Python ci-après présente la fonction récursive `chargement_glouton` dont les paramètres sont :

- `liste` : une liste de colis triées par poids décroissants ;
- `rang` : un indice compris entre 0 inclus et `len(liste)` inclus ;
- `charge` : une charge exprimée en kilogrammes ;

et qui renvoie la liste des colis à charger en appliquant l'algorithme glouton, en supposant que la charge restante dans le camion est `capacité`, et en ne considérant que les colis de `liste` d'indice supérieur ou égal à `rang`.

```
1 def chargement_glouton(liste, rang, capacite):
2     if rang == len(liste):
3         return ...
4     elif liste[rang].poids <= ...:
5         return ... + chargement_glouton(liste, ..., ...)
6     else:
7         return chargement_glouton(liste, ..., ...)
```

8. Recopier et compléter le code ci-dessus de la fonction `chargement_glouton`.

9. Expliquer brièvement pourquoi, lors d'un appel à la fonction `chargement_glouton`, on peut obtenir l'erreur suivante.

   `RecursionError: maximum recursion depth exceeded while calling a Python object`.

10. Écrire une fonction `chargement_glouton2` **itérative** (sans récursivité) qui prend en paramètres `liste` une liste de colis triés par poids décroissants et `capacite` la capacité du camion exprimée en kilogrammes, et qui renvoie la liste des colis à charger pour maximiser le poids total sans dépasser la capacité.

    On pourra créer une liste `colis_a_charger`, puis parcourir les colis triés en les ajoutant à cette liste tant que le poids total n'excède pas la capacité du camion.

Exercice 3 (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.

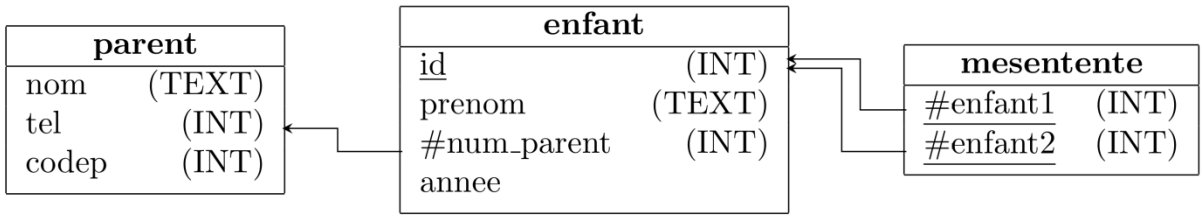


Figure 1. Schéma relationnel de la BDD

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**.
2. Expliquer quelle contrainte de domaine supplémentaire serait pertinente pour cet attribut `annee`.
3. Donner un exemple d’attribut de la table **enfant** qui suit une contrainte de référence.
4. En expliquant ce choix, proposer une clef primaire pour la table **parent**.

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.

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 = ...;
```

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

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

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

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

Partie B : graphes et algorithmique

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.

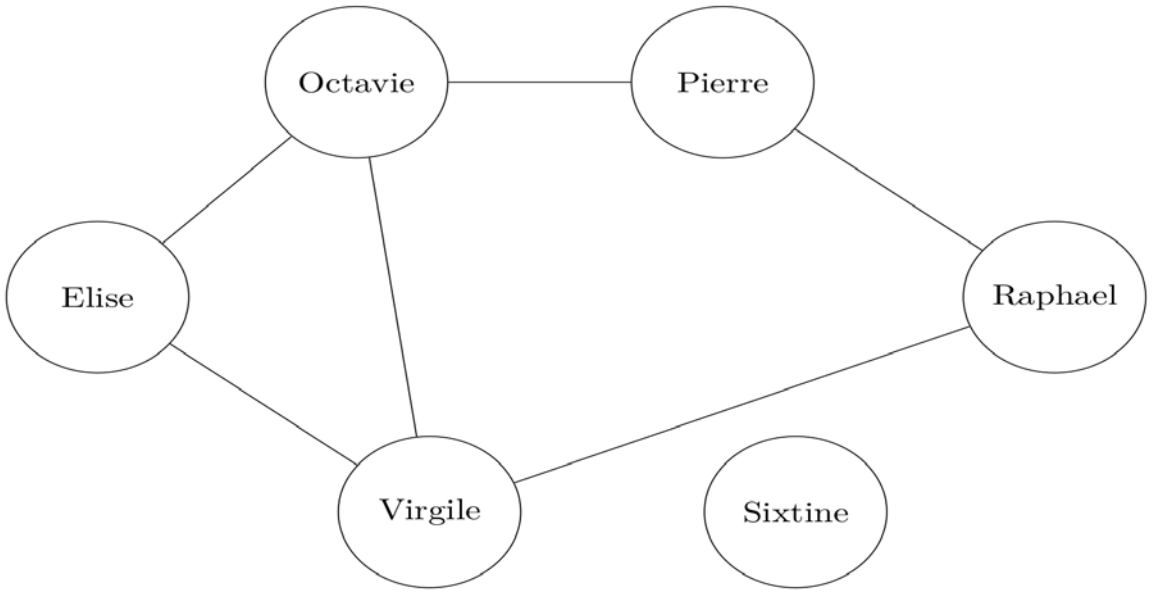


Figure 2. Graphe `g1`

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

```
1 g1 = {'Elise': ['Octavie', 'Virgile'],
2       'Octavie': ['Elise', 'Pierre', 'Virgile'],
3       'Pierre': ['Octavie', 'Raphael'],
4       'Raphael': ['Pierre', 'Virgile'],
5       'Sixtine': [],
6       'Virgile': ['Elise', 'Octavie', 'Raphael']
7      }
```

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

11. Dessiner le graphe g2 défini ci-dessous.

```
1 g2 = {'Adrien': ['Elisabeth', 'Lea'],
2       'Elisabeth': ['Adrien', 'Ian', 'Luca'],
3       'Ian': ['Elisabeth', 'Joseph', 'Luca'],
4       'Joseph': ['Ian'],
5       'Lea': ['Adrien'],
6       'Luca': ['Elisabeth', 'Ian']
7      }
```

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.

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 ... and ...:
 8             sommets[...] = sommets[...]
 9             j = j - 1
10         ...
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.

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.

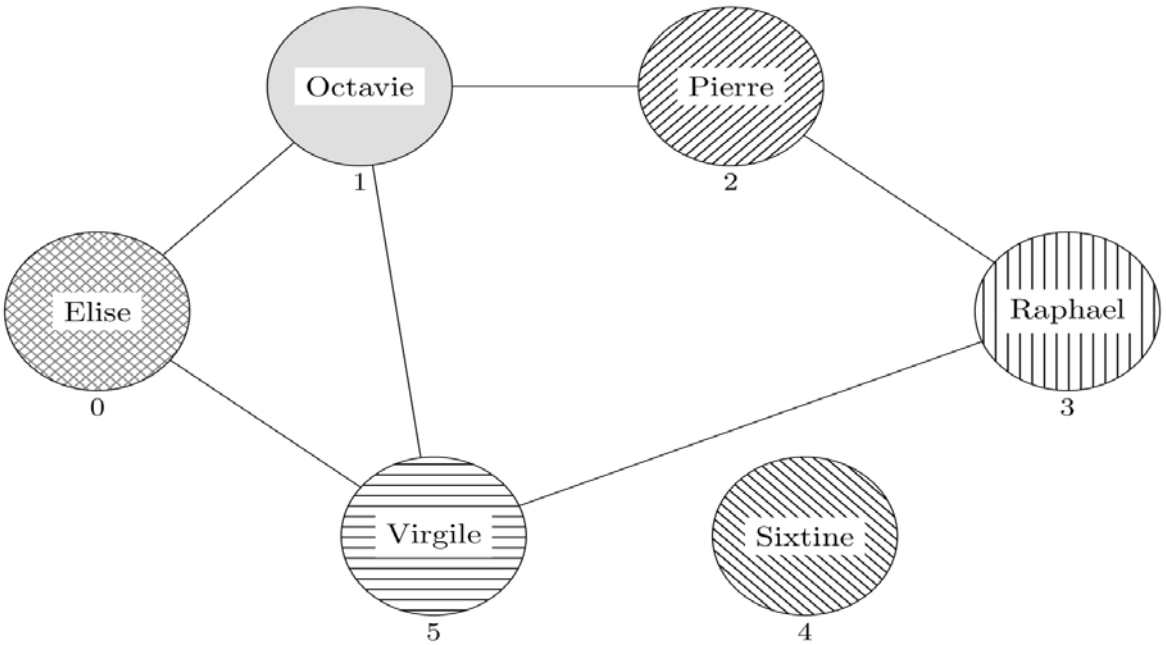


Figure 3. Graphe g1 coloré

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).

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).

```
 1 def couleurs_voisins(g, dc, s):
 2     return [dc[v] for v in g[s]]
 3
 4 def plus_petite_couleur_hors_voisins(g, dc, s):
 5     couleur = 0
 6     n = len(g)
 7     cvoisins = couleurs_voisins(g, dc, s)
 8     while couleur < n:
 9         if couleur not in cvoisins:
10             return couleur
11         couleur = couleur + 1
12     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.

```
1 def colorer_graphe(g, dc):
2     # Pré-condition : les clés de dc sont les sommets
3     # de g, et les valeurs de dc sont toutes à -1
4     for s in dc:
5         couleur = ...
6         ... = 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.

```
1 def welsh_powell(g):
2     # initialisation à -1 pour tous les sommets dans le
dictionnaire dc
3     dc = ... # possiblement plusieurs lignes
4     # coloration en suivant l'approche de Welsh-Powell
5     for ...
6         ...
7         ...
8     return dc
```