2.1 Programmation Orienté Objet : BAC

1 Polynésie J1 - 2022

Exercice 1

Cet exercice traite du thème « programmation », et principalement de la récursivité.

On rappelle qu'une chaîne de caractères peut être représentée en Python par un texte entre guillemets "" et que :

- la fonction `len` renvoie la longueur de la chaîne de caractères passée en paramètre ;
- si une variable `ch` désigne une chaîne de caractères, alors `ch[0]` renvoie son premier caractère, `ch[1]` le deuxième, ... ;
- l'opérateur `+` permet de concaténer deux chaînes de caractères.

Exemples :

```
>>> texte = "bricot"
>>> len(texte)
6
>>> texte[0]
"b"
>>> texte[1]
"r"
>>> "a" + texte
"abricot"
```

On s'intéresse dans cet exercice à la construction de chaînes de caractères suivant certaines règles de construction.

Règle A : une chaîne est construite suivant la règle A dans les deux cas suivants:

- soit elle est égale à `"a"` ;
- soit elle est de la forme `"a"+chaine+"a"`, où `chaine` est une chaîne de caractères construite suivant la règle A.

Règle B : une chaîne est construite suivant la règle B dans les deux cas suivants :

- soit elle est de la forme `"b"+chaine+"b"`, où `chaine` est une chaîne de caractères construite suivant la règle A ;
- soit elle est de la forme `"b"+chaine+"b"`, où `chaine` est une chaîne de caractères construite suivant la règle B.

On a reproduit ci-dessous l'aide de la fonction `choice` du module `random`.

```
>>> from random import choice
>>> help(choice)
Help on method choice in module random:

choice(seq) method of random.Random instance
    Choose a random element from a non-empty sequence.
```

La fonction `A()` ci-dessous renvoie une chaîne de caractères construite suivant la règle A, en choisissant aléatoirement entre les deux cas de figure de cette règle.

```
def A():
    if choice([True, False]):
        return "a"
    else:
        return "a" + A() + "a"
```

Q1. **(a)** Cette fonction est-elle récursive ? Justifier.

(b) La fonction `choice([True, False])` peut renvoyer `False` un très grand nombre de fois consécutives. Expliquer pourquoi ce cas de figure amènerait à une erreur d'exécution.

Dans la suite, on considère une deuxième version de la fonction `A`. À présent, la fonction prend en paramètre un entier `n` tel que, si la valeur de `n` est négative ou nulle, la fonction renvoie `"a"`. Si la valeur de `n` est strictement positive, elle renvoie une chaîne de caractères construite suivant la règle A avec un `n` décrémenté de 1, en choisissant aléatoirement entre les deux cas de figure de cette règle.

```
def A(n):
    if ... or choice([True, False]) :
        return "a"
    else:
        return "a" + ... + "a"
```

Q2. **(a)** Recopier sur la copie et compléter aux emplacements des points de suspension le code de cette nouvelle fonction `A`.

(b) Justifier le fait qu'un appel de la forme `A(n)` avec `n` un nombre entier positif inférieur à 50, termine toujours.

On donne ci-après le code de la fonction récursive B qui prend en paramètre un entier `n` et qui renvoie une chaîne de caractères construite suivant la règle B.

```
def B(n):
    if n <= 0 or choice([True, False]):
        return "b" + A(n-1) + "b"
    else:
        return "b" + B(n-1) + "b"
```

On admet que :

- les appels `A(-1)` et `A(0)` renvoient la chaîne `"a"` ;
- l'appel `A(1)` renvoie la chaîne `"a"` ou la chaîne `"aaa"` ;
- l'appel `A(2)` renvoie la chaîne `"a"`, la chaîne `"aaa"` ou la chaîne `"aaaaa"`.

Q3. Donner toutes les chaînes possibles renvoyées par les appels `B(0)`, `B(1)` et `B(2)`.

On suppose maintenant qu'on dispose d'une fonction `raccourcir` qui prend comme paramètre une chaîne de caractères de longueur supérieure ou égale à 2, et renvoie la chaîne de caractères obtenue à partir de la chaîne initiale en lui ôtant le premier et le dernier caractère.
Par exemple :

```
>>> raccourcir("abricot")
"brico"
>>> raccourcir("ab")
""
```

Q4. **(a)** Recopier sur la copie et compléter les points de suspension du code de la fonction `regleA` ci-dessous pour qu'elle renvoie `True` si la chaîne passée en paramètre est construite suivant la règle A, et `False` sinon.

```
def regleA(chaine):
    n = len(chaine)
    if n >= 2:
        return chaine[0] == "a" and chaine[n-1] == "a" and regleA(...)
    else:
        return chaine == ...
```

(b) Écrire le code d'une fonction `regleB`, prenant en paramètre une chaîne de caractères et renvoyant `True` si la chaîne est construite suivant la règle B, et `False` sinon.

2 Réunion J1 - 2023

Exercice 2

Cet exercice porte sur la programmation orientée objet et les dictionnaires.

Dans le tableau ci-dessous, on donne les caractéristiques nutritionnelles, pour une quantité de 100 g, de quelques aliments.

	Lait entier UHT	Farine de blé	Huile de tournesol
Énergie (kcal)	65.1	343	900
Protéines (grammes)	3.32	11.7	0
Glucides (grammes)	4.85	69.3	0
Lipides (grammes)	3.63	0.8	100

Figure 1 : Caractéristiques nutritionnelles

Pour chaque aliment, on souhaite stocker les informations dans un objet de la classe `Aliment` définie ci-dessous, où `e`, `p`, `g` et `l` sont de type `float` et désignent respectivement les quantités d'énergie, de protéines, de glucides et de lipides de l'aliment.

```
class Aliment:
    def __init__(self, e, p, g, l):
        self.energie = e
        self.proteines = p
        self.glucides = g
        self.lipides = l
```

Q1. **(a)** Écrire, à l'aide du tableau des caractéristiques nutritionnelles de la Figure 1, l'instruction en langage python pour instancier l'objet `lait`.

(b) Donner l'instruction qui permet d'obtenir la valeur `65.1` de l'objet `lait` instancié dans la question précédente.

Une erreur s'est introduite dans le tableau de la Figure 1 : la masse de protéines dans le lait est `3.4` au lieu de `3.32`.

(c) Donner l'instruction qui modifie la masse de protéines de l'objet `lait` instancié dans la question **Q1 (a)**.

On souhaite ajouter une méthode `energie_reelle` à la classe `Aliment` qui calcule l'énergie en kcal d'un aliment en fonction d'une masse donnée.

Par exemple :
pour 245 grammes de lait, l'énergie réelle sera $245 \times 65.1 \div 100 = 159.495$ kcal.
L'instruction `lait.energie_reelle(245)` renvoie alors `159.495`

Q2. Recopier et compléter les lignes 1 et 2 dans la méthode ci-dessous.

```
1    def energie_reelle(..., masse):
2        return ...
```

Q3. On regroupe les caractéristiques nutritionnelles du tableau de la Figure 1 dans le dictionnaire suivant, les clés étant des chaînes de caractères donnant le nom de l'aliment et les valeurs associées des objets de la classe `Aliment` :

```
nutrition = {'lait' : Aliment(65.1,3.4,4.85,3.63),
             'farine' : Aliment(343,11.7,69.3,0.8),
             'huile' : Aliment(900,0,0,100)
             }
```

a) Donner l'instruction qui permet d'obtenir la valeur énergétique en kcal du lait à partir des données de ce dictionnaire.

b) Donner l'instruction qui permet d'obtenir la valeur énergétique réelle de 220 grammes de lait à partir des données de ce dictionnaire.

Une recette de gâteau (sans œuf) utilise les ingrédients suivants :

- 230 g de farine ;
- 220 g de lait ;
- 100 g d'huile.

Les quantités d'ingrédients, en grammes, sont regroupées dans le dictionnaire suivant :
`recette_gateau = {'lait ' : 220, 'farine' :230, 'huile':100}`

Q4. Écrire, en utilisant la classe `Aliment` et la méthode `energie_reelle`, les instructions nécessaires pour calculer l'énergie réelle totale du gâteau.

3 France J1 - 2023

Exercice 3

Cet exercice traite de programmation orientée objet en python *et d'algorithmique.*

Un pays est composé de différentes régions. Deux régions sont voisines si elles ont au moins une frontière en commun. L'objectif est d'attribuer une couleur à chaque région sur la carte du pays sans que deux régions voisines aient la même couleur et en utilisant le moins de couleurs possibles.
La figure 1 ci -dessous donne un exemple de résultat de coloration des régions de la France métropolitaine.

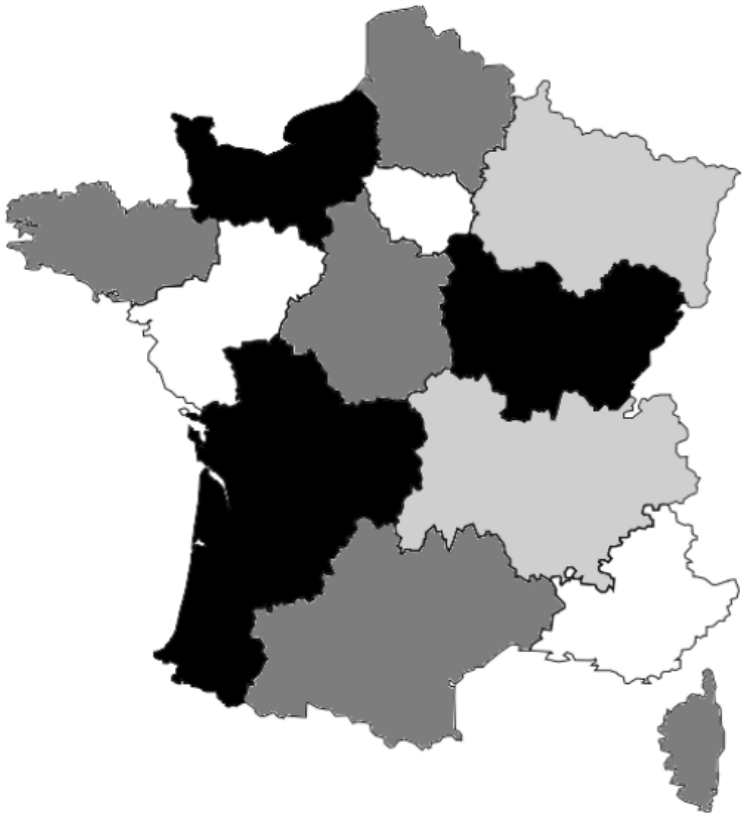


Figure 1 - Carte coloriée des régions de France métropolitaine

On rappelle quelques fonctions et méthodes des tableaux (le type `list` en python) qui pourront être utilisées dans cet exercice :

- `len(tab)` : renvoie le nombre d'éléments du tableau `tab` ;
- `tab.append(elt)` : ajoute l'élément `elt` en fin de tableau `tab` ;
- `tab.remove(elt)` : enlève la première occurrence de `elt` de `tab` si `elt` est dans `tab`. Provoque une erreur sinon.

Exemple :

- `len([1, 3, 12, 24, 3])` renvoie `5` ;

— avec `tab = [1, 3, 12, 24, 3]`, l'instruction `tab.append(7)` modifie `tab` en `[1, 3, 12, 24, 3, 7]` ;

— avec `tab = [1, 3, 12, 24, 3]`, l'instruction `tab.remove(3)` modifie `tab` en `[1, 12, 24, 3]`.

Les deux parties de cet exercice forment un ensemble. Cependant, il n'est pas nécessaire d'avoir répondu à une question pour aborder la suivante. En particulier, on pourra utiliser les méthodes des questions précédentes même quand elles n'ont pas été codées.

Pour chaque question, toute trace de réflexion sera prise en compte.

Partie 1

On considère la classe `Region` qui modélise une région sur une carte et dont le début de l'implémentation est :

```
class Region:
    '''Modélise une région d'un pays surune carte.'''
    def __init__(self, nom_region):
        '''
        initialse une région
        : param nom_region (str) le nom de la région
        '''
        self.nom = nom_region
        # tableau des regions voisines, vide au départ
        self.tab_voisines = []
        # tableau des couleurs disponibles pour colorier la région
        self.tab_couleurs_disponibles = ['rouge', 'vert', 'bleu', 'jaune',
                                         'orange', 'marron']
        # couleur attribuée à la région et non encore choisie au départ
        self.couleur_attribuee = None
```

Q1. Associer, en vous appuyant sur l'extrait de code précédent, les noms `nom`, `tab_voisines`, `tab_couleurs_disponibles` et `couleur_attribuee` au terme qui leur correspond parmi : *objet*, *attribut*, *méthode* ou *classe*.

Q2. Indiquer le type du paramètre `nom_region` de la méthode `__init__` de la classe `Region`.

Q3. Donner une instruction permettant de créer une instance nommée `ge` de la classe `Region` correspondant à la région dont le nom est « Grand Est ».

Q4. Recopier et compléter la ligne 7 de la méthode de la classe Region ci-dessous :

```
1   def renvoie_premiere_couleur_disponible(self):
2       '''
3       Renvoie la première couleur du tableau des couleurs
4       disponibles supposé non vide.
5       : return (str)
6       '''
7       return ...
```

Q5. Recopier et compléter la ligne 6 de la méthode de la classe `Region` ci-dessous :

```
1   def renvoie_nb_voisines(self) :
2       '''
3       Renvoie le nombre de régions voisines.
4       : return (int)
5       '''
6       return ...
```

Q6. Compléter la méthode de la classe `Region` ci-dessous à partir de la ligne 7 :

```
    def est_coloriee(self):
        '''
        Renvoie True si une couleur a été attribuée à cette
        région et False sinon.
        : return (bool)
        '''
        ...
```

Q7. Compléter la méthode de la classe `Region` ci-dessous à partir de la ligne 9 :

```
    def retire_couleur(self, couleur):
        '''
        Retire couleur du tableau de couleurs disponibles de la région
        si elle est dans ce tableau. Ne fait rien sinon.
        : param couleur (str)
        : ne renvoie rien
        : effet de bord sur le tableau des couleurs disponibles
        '''
        ...
```

Q8. Compléter la méthode de la classe `Region` ci-dessous, à partir de la ligne 7, en utilisant une boucle :

```
    def renvoie_premiere_couleur_disponible(self):
        '''
        Renvoie la première couleur du tableau des couleurs
        disponibles supposé non vide.
        : return (str)
        '''
        return ...
```

Partie 2

Dans cette partie :

- on considère qu'on dispose d'un ensemble d'instances de la classe `Region` pour lesquelles l'attribut `tab_voisines` a été renseigné ;
- on pourra utiliser les méthodes de la classe `Region` évoquées dans les questions de la partie 1 :
  - `renvoie_premiere_couleur_disponible`
  - `renvoie_nb_voisines`
  - `est_coloriee`
  - `retire_couleur`
  - `est_voisine`

On a créé une classe `Pays` :

- cette classe modélise la carte d'un pays composé de régions ;
- l'unique attribut `tab_regions` de cette classe est un tableau (type `list` en `python`) dont les éléments sont des instances de la classe `Region`.

Q9. Recopier et compléter la méthode de la classe `Pays` ci-dessous à partir de la ligne 7 :

```
    def renvoie_tab_regions_non_coloriees(self):
        '''
        Renvoie un tableau dont les éléments sont les régions
        du pays sans couleur attribuée.
        : return (list) tableau d'instances de la classe Region
        '''
        ...
```

Q10. On considère la méthode de la classe `Pays` ci-dessous.

```python
    def renvoie_max(self):
        nb_voisines_max = -1
        region_max = None
        for reg in self.renvoie_tab_regions_non_coloriees():
            if reg.renvoie_nb_voisines() > nb_voisines_max:
                nb_voisines_max = reg.renvoie_nb_voisines()
                region_max = reg
        return region_max
```

(a) Expliquer dans quel cas cette méthode renvoie `None`.

(b) Indiquer, dans le cas où cette méthode ne renvoie pas `None`, les deux particularités de la région renvoyée.

Q11. Coder la méthode `colorie(self)` de la classe `Pays` qui choisit une couleur pour chaque région du pays de la façon suivante :

- On récupère la région non coloriée qui possède le plus de voisines.
- Tant que cette région existe :
  - La couleur attribuée à cette région est la première couleur disponible dans son tableau de couleurs disponibles.
  - Pour chaque région voisine de la région :
    - si la couleur choisie est présente dans le tableau des couleurs disponibles de la région voisine alors on la retire de ce tableau.
  - On récupère à nouveau la région non coloriée qui possède le plus de voisines.

4 Nouvelle-Calédonie J2 - 2022

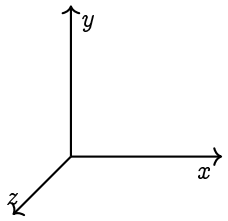
Exercice 4

Cet exercice porte sur de l'algorithmique et de la programmation en langage Python. Il aborde la programmation orientée objet.

L'objectif de cet exercice est de créer un jeu vidéo. Il s'agit d'un jeu de plateau sur le thème des chevaliers de la table ronde dans lequel plusieurs personnages doivent réaliser des missions. Ces personnages doivent récupérer des objets sur leur parcours. La récupération de ces objets leur permettra de gagner des points de vie ou d'en perdre. Dans l'environnement du jeu, les héros vont rencontrer des personnages qui pourraient se révéler dangereux et contre lesquels ils devront parfois engager des batailles.

Partie 1

On s'intéresse ici au déplacement des personnages. L'espace dans lequel ils évoluent est représenté par un repère orthonormé à trois axes. La position de chaque personnage sera repérée par ses attributs x, y, z.



1. (a) Recopier sur la copie et compléter le constructeur de la classe Personnage positionnant un personnage aux coordonnées choisies x, y et z.

```python
class Personnage:
    def __init__(self, coordx, coordy, coordz):
        ...
        ...
        ...
```

(b) Écrire une méthode `avancex` de la classe `Personnage` permettant d'augmenter d'une unité la coordonnée x d'un personnage.
(c) Écrire une méthode `raz` permettant de mettre toutes les coordonnées d'un personnage à zéro.
(d) Écrire une méthode `coord` renvoyant les coordonnées d'un personnage sous forme d'un tuple.

2. En utilisant les méthodes écrites dans la partie 1, écrire des lignes de code permettant :
   (a) de créer un personnage arthur démarrant à la position (5,5,5) ;
   (b) d'augmenter d'une unité la coordonnée x du personnage arthur ;
   (c) d'afficher les coordonnées du personnage arthur.

Partie 2

Chaque personnage du jeu va rencontrer différentes situations (potions, pièges, ...) qui vont faire évoluer ses points de vie. La classe Personnage est modifiée afin de permettre ces évolutions.

```
1  import random
2  class Personnage :
3      def __init__(self, coordx, coordy, coordz, point_de_vie):
4          ... # défini dans la partie 1
5          self.vie = point_de_vie
6
7      def avancex ... # défini dans la partie 1
8      def raz ... # défini dans la partie 1
9      def coord ... # défini dans la partie 1
10
11     def get_etat(self) :
12         return self.vie
13
14     def newgame(self) :
15         ... # défini dans la partie 2
16
17     def potionmystere(self) :
18         if random.randint(1,2) == 1 :
19             nbPoint = -1
20         else :
21             nbPoint = +1
22         self.vie = self.vie + nbPoint
23
24     def piege(self) :
25         self.vie = self.vie - 10
26
27     def repos(self) :
28         self.vie = self.vie + 5
```

1. Indiquer les valeurs possibles de la nouvelle variable `valeurMerlin` après exécution du programme ci-dessous.

```
1  merlin = Personnage (4,5,8,15)
2  merlin.potionmystere()
3  valeurMerlin = merlin.get_etat()
```

2. Indiquer la valeur de la nouvelle variable `valeurMerlin` après exécution du programme ci-dessous.

```
1  merlin = Personnage (4,5,8,20)
2  merlin.piege()
3  merlin.piege()
4  valeurMerlin = merlin.get_etat()
```

3. Écrire sur la copie, la méthode `newgame` permettant, si le nombre de points de vie est inférieur ou égal à 0, de :
   (a) ramener les coordonnées du personnage à (0,0,0),
   (b) lui attribuer 15 points de vie.

   Pour intégrer les combats au jeu, on modifie et complète la classe Personnage en rajoutant les méthodes suivantes :

```python
def perdre_vie(self,points) :
    self.vie = self.vie - points
    self.newgame()

def attaquer(self, autre) :
    autre.perdre_vie(self.degats)
```

4. Écrire un programme en langage Python permettant :
   (a) la création d'une instance `lancelot` de cette classe ayant pour attributs 5,5,5 en coordonnées, 15 en `point_de_vie` et 3 en `point_degats`.
   (b) la création d'une instance sorcier de cette classe ayant pour attributs 6,5,5 en coordonnées, 15 en `point_de_vie` et 2 en `point_degats`
   (c) à `lancelot` d'attaquer le sorcier une première fois.
   (d) au sorcier d'attaquer à son tour lancelot.
   (e) à lancelot de répliquer en attaquant quatre fois de suite le sorcier.
   (f) d'afficher le nombre de points de vie de lancelot et du sorcier.

5 Nouvelle-Calédonie J1 - 2022

Exercice 5

Cet exercice traite de programmation orientée objet en python *et d'algorithmique.*

Des élus d'un canton français décident de mettre en place une monnaie locale complémentaire dans leur circonscription, appelée le « sou ». L'objectif est d'encourager la population à acheter auprès de vendeurs et producteurs locaux.
La plus petite valeur du sou est le billet de 1 sou, il ne peut donc pas être séparé en dessous de ce montant. Le sou a un taux de change à parité avec l'euro (1 sou = 1 euro), de façon à ce que le prix d'un article puisse être intégralement payé en sous. Si le prix d'un article n'est pas entier, la différence peut être payée en euros. Par exemple, un article coûtant 3,50 € peut être payé avec 3 sous et 50 centimes d'euros.
Le canton a créé une association chargée de gérer les comptes de ses futurs adhérents au moyen d'une application en langage Python. On a commencé à créer une classe Compte dont les propriétés sont : le numéro de compte, le nom de l'adhérent, son adresse et le solde du compte. Lors de la création d'un compte, il faudra renseigner toutes les propriétés sauf le solde qui sera mis à 0 sou. Les méthodes de cette classe sont les suivantes :

Méthodes de la classe Compte	Description
`__init__(self,numero,adherent,adresse)`	Crée un nouveau compte et met le solde à 0
`crediter(self,montant)`	Ajoute montant à solde
`debiter(self, montant)`	Enlève montant à solde
`est_positif(self)`	Renvoie Vrai si le solde du compte est positif ou nul

Partie 1 : Création de la classe compte

On a commencé à écrire la classe Compte.

```python
class Compte:
    def __init__(self, numero, adherent, adresse):
        self.numero = numero
        self.adherent = adherent
        self.adresse = adresse
        self.solde = 0
```

1. Écrire sur la copie la méthode `crediter`.
2. Écrire sur la copie la méthode `debiter`.
3. Écrire sur la copie la méthode `est_positif`.

Partie 2 : Utilisation de la classe compte

Monsieur Martin souhaite rejoindre la communauté des utilisateurs du sou et déposer 200 € sur son compte en sou.

1. Écrire la ligne de code en langage Python permettant de créer le compte `cpt_0123456A` dont le numéro est "0123456A", le titulaire est "MARTIN Dominique" qui habite à l'adresse "12 rue des sports".
2. Écrire la ligne de code en langage Python permettant de déposer 200 € sur le compte de monsieur Martin.
3. Monsieur Martin souhaite transférer 50 sous de son compte « `cpt_0123456A` » vers un autre compte « `cpt_2468794E` ». On a besoin pour cela d'ajouter une méthode à la classe Compte.
   Écrire la méthode `transferer(self, autre_compte, montant)` qui transfère le montant passé en paramètre vers un autre compte. On suppose que le compte courant est suffisamment approvisionné. On utilisera les autres méthodes de la classe Compte sans en modifier directement les attributs.

Partie 3 : Gestion des comptes

Pour en faciliter la gestion, on crée une liste Python `liste_comptes` contenant tous les comptes des adhérents. Cette liste sera donc composée d'objets de type compte. Le gestionnaire de l'association souhaite relancer les adhérents dont le compte est débiteur, c'est-à-dire dont le solde est négatif. Il faut, pour cela, ajouter une autre fonction au programme en langage Python.

1. Écrire la fonction `rechercher_debiteurs` qui prend en argument `liste_comptes` et renvoie une liste de dictionnaires dont la première clé est le nom de l'adhérent et la seconde clé le solde de son compte en négatif.
   Exemple de résultat :
   ```
   liste_debiteurs = [{'Nom': 'DUPONT Thomas', 'solde': -60}, {'Nom': 'CARNEIRO Sarah', 'solde': -
   ```
2. Écrire la fonction `urgent_debiteur` qui prend en argument la liste de dictionnaires et renvoie le nom de l'adhérent le plus endetté. On suppose qu'aucun adhérent n'a la même dette.