Asie - juin 2024 - sujet 1

Exercice 1 (Programmation Python, POO et algorithmique - 6 points)
Une entreprise doit placer des antennes relais le long d'une rue rectiligne. Une antenne relais de portée (ou rayon) p couvre toutes les maisons qui sont à une distance inférieure ou égale à p de l'antenne.

Connaissant les positions des maisons dans la rue, l'objectif est de placer les antennes le long de la rue, pour que toutes les maisons soient couvertes, tout en en minimisant le nombre d'antennes utilisées.

La rue est représentée par un axe, et les maisons sont représentées des points sur cet axe :

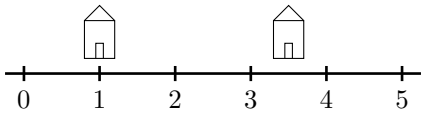

FIGURE 1 – Deux maisons sur une rue, repérée par leur abscisse 1 et 3,5

Les entités manipulées sont modélisées en utilisant la programmation orientée objet.

```
class Maison:
    def __init__(self, position):
        self._position = position

    def get_pos_maison(self):
        return self._position

class Antenne:
    def __init__(self, position, rayon):
        self._position = position
        self._rayon = rayon

    def get_pos_antenne(self):
        return self._position

    def get_rayon(self):
        return self._rayon
```

1. Donner le code qui crée et initialise deux variables `m1` et `m2` avec des instances de la classe `Maison` situées aux abscisses 1 et 3,5 (Figure 1).

On ajoute à présent une antenne ayant un rayon d'action de 1 à la position 2,5 :

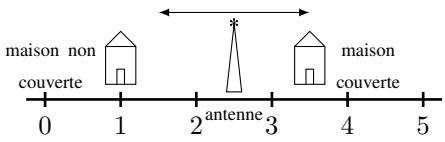

FIGURE 2 – L'antenne placée en 2,5 et de rayon d'action 1 couvre la maison en 3,5 mais pas celle en 1

2. Donner le code qui crée la variable `a` correspondant à l'antenne à la position 2,5 avec le rayon d'action 1.

On souhaite modéliser une rue par une liste d'objets de type `Maison`. Cette liste sera construite à partir d'une autre liste contenant des nombres correspondant aux positions des maisons. La fonction `creation_rue` réalise ce travail. Elle prend en paramètre une liste de positions et renvoie une liste d'objets de type `Maison`.

3. Recopier le schéma ci-dessous et le compléter pour donner une représentation graphique de la situation créée par l'instruction `creation_rue([0, 2, 3, 4, 5, 7, 9, 10.5, 11.5])`

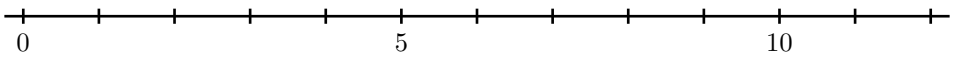

FIGURE 3 – Axe pour représenter le problème avec 9 maisons

4. Compléter le code donné ci-dessous de la fonction `creation_rue`.

```
1 def creation_rue(pos):
2     pos.sort()
3     maisons = []
4     for p in pos:
5         m = Maison(p)
6         maisons.append(...)
7     return ...
```

Pour rappel, la commande `tab.sort()` trie la liste `tab`.

La méthode `couvre` de la classe `Antenne` prend en paramètre un objet de type `Maison` et indique par un booléen si l'antenne couvre la maison en question ou non. La méthode peut être utilisée ainsi (en supposant que les objets précédents `m1`, `m2` et `a` existent) :

```
>>> a.couvre(m1)
False
>>> a.couvre(m2)
True
```

5. Compléter la fonction `couvre` ci-dessous, en veillant à ne pas accéder directement aux attributs d'une maison depuis la classe `Antenne` (on pourra utiliser les méthodes `get_pos_maison` , `get_pos_antenne` et `get_rayon`).
Pour rappel, la fonction valeur absolue se nomme `abs()` en Python.

```
1 # Méthode à ajouter dans la classe Antenne
2 def couvre(self, maison):
3     # Code à compléter (éventuellement plusieurs lignes)
4     ...
```

La fonction `strategie_1` est donnée ci-dessous. L'objectif est de placer des antennes dans une rue. Elle est fournie à la société qui place les antennes. La fonction prend en paramètre une liste d'objets de type `Maison` (qu'on supposera triée par abscisse croissante) et le rayon d'action des antennes (`float`). Cette fonction renvoie une liste d'objets de type `Antenne` ayant ce rayon d'action et couvrant toutes les maisons de la rue.

```
1 def strategie_1(maisons, rayon):
2     ''' Prend en paramètre une liste de maisons et le rayon
3         d'action des antennes et renvoie une liste d'antennes
4     '''
5     antennes = [Antenne(maisons[0].get_pos_maison(), rayon)]
6     for m in maisons[1:]:
7         if not antennes[-1].couvre(m):
8             antennes.append(Antenne(m.get_pos_maison(), rayon))
9     return antennes
```

Pour rappel :

- `tab[1:]` correspond aux éléments de `tab` à partir de l'indice `1` jusqu'à la fin de la liste ;
- `tab[-1]` correspond au dernier élément de la liste `tab`.

6. Indiquer ce que renvoie cette suite d'instructions après exécution :

```
>>> maisons = creation_rue([0, 2, 3, 4, 5, 7, 9, 10.5, 11.5])
>>> antennes = strategie_1(maisons, 2)
>>> print([a.get_pos_antenne() for a in antennes])
```

Une amélioration est possible et la société qui pose les antennes souhaite implémenter l'algorithme suivant :

- considérer les maisons dans l'ordre des abscisses croissantes ;
- dès qu'une maison n'est pas couverte, placer une antenne à la plus grande abscisse telle qu'elle couvre cette maison. Par exemple, si la maison d'abscisse 5 est la première maison non couverte, alors, on placera l'antenne en $5 + r$ si r est le rayon d'action de l'antenne.

7. On considère la rue composée des maisons situées aux abscisses `[0, 2, 3, 4, 5, 7, 9, 10.5, 11.5]`. Recopier et compléter le schéma ci-dessous en indiquant l'emplacement des antennes selon cette nouvelle stratégie. On suppose que le rayon d'action est toujours 2.

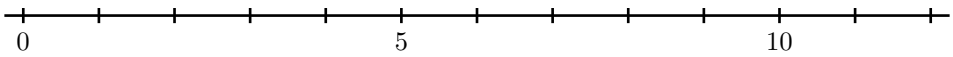

FIGURE 4 – Axe pour représenter le résultat de la nouvelle stratégie

8. Cet algorithme étant *a priori* plus économe en antennes, proposer une fonction `strategie_2`, sur le modèle de `strategie_1` qui implémente cette nouvelle stratégie.
9. Comparer le coût en nombre d'opérations des deux stratégies en fonction du nombre `n` de maisons dans la rue. On admet que le coût de la fonction `append` est constant.

Exercice 2 (Graphes, programmation Python et piles - 6 points)
On s'intéresse à la fabrication de pain. La recette est fournie sous la forme de tâches à réaliser. Cette recette est réalisée par une personne seule.

(a) Préparer 500g de farine.
(b) Préparer 1/3 de litre d'eau (33cl).
(c) Préparer 1 c. à café de sel.
(d) Préparer 20g de levure de boulanger.
(e) Faire tiédir l'eau dans une casserole.
(f) Délayer la levure dans l'eau tiède.
(g) Laisser reposer la levure 5 minutes.
(h) Préparer un grand saladier.
(i) Verser la farine dans le saladier.
(j) Verser le sel dans le saladier.
(k) Mélanger la farine et le sel puis creuser un puits.
(l) Verser l'eau mélangée à la levure dans le puits.
(m) Pétrir jusqu'à obtenir une pâte homogène.
(n) Couvrir à l'aide d'un linge humide et laisser fermenter au moins 1h30.
(o) Disposer dans le fond du four un petit récipient contenant de l'eau.
(p) Préchauffer un four à 200 degrés Celsius.
(q) Fariner un plan de travail.
(r) Verser la pâte à pain sur le plan de travail.
(s) Pétrir rapidement la pâte à pain.
(t) Disposer la pâte dans un moule à cake.
(u) Mettre au four pour 15 à 20 minutes, arrêter le four et sortir le pain.

La figure 1 représente les différentes tâches et les dépendances entre ces tâches sous la forme d'un graphe. Chaque sommet du graphe représente une tâche à réaliser. Les dépendances entre les tâches sont représentées par les arcs entre les sommets.

Par exemple, il y a une flèche sur l'arc qui part du sommet d'étiquette (l) et qui atteint le sommet d'étiquette (m) car il faut avoir réalisé la tâche « Verser l'eau mélangée à la levure dans le puits. » (l) avant de pouvoir réaliser la tâche « Pétrir jusqu'à obtenir une pâte homogène. » (m).

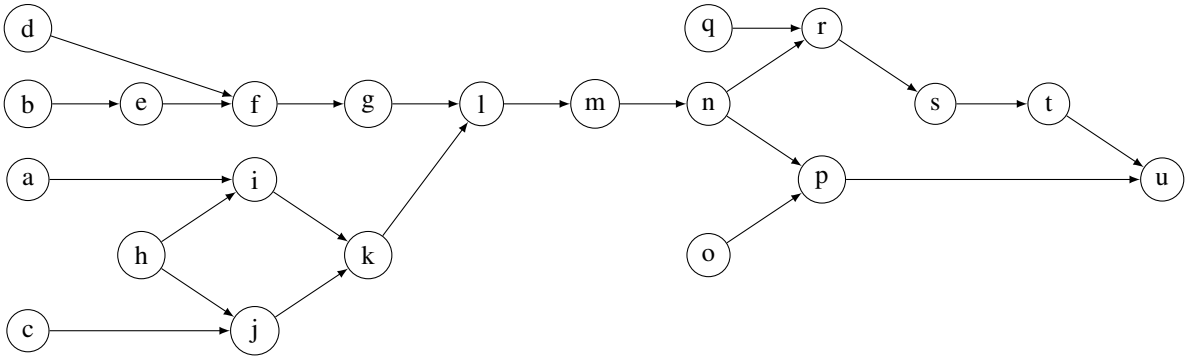

FIGURE 1 – Recette du pain : tâches à effectuer avec leurs dépendances

1. Dire, sans justifier, s'il s'agit d'un graphe orienté ou non orienté.
2. D'après le graphe, dire s'il est possible d'effectuer les réalisations dans chacun des ordres suivants :
   - réaliser la tâche (f) puis la tâche (g)
   - réaliser la tâche (g) puis la tâche (f)
   - réaliser la tâche (i) puis la tâche (j)
   - réaliser la tâche (j) puis la tâche (i)

3. Donner toutes les tâches qu'il faut nécessairement avoir réalisées depuis le début pour pouvoir réaliser la tâche (k). Ne donner que les tâches nécessaires.
4. Indiquer, sans justifier, si le graphe de la figure 1 contient un cycle.

Graphe de tâches

On s'intéresse désormais de manière plus générale à un graphe de tâches avec des dépendances.

Les sommets sont nommés par des indices. Comme précédemment, un arc orienté d'un sommet d'indice `i` à un sommet d'indice `j` signifie que la tâche représentée par le sommet d'indice `i` doit être réalisée avant la tâche représentée par le sommet d'indice `j`.

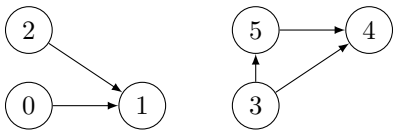

FIGURE 2 – Exemple de graphe de dépendances entre 6 tâches

5. Déterminer un ordre permettant de réaliser toutes les tâches représentées dans le graphe de la figure 2 en respectant les dépendances entre les tâches.

Voici une matrice d'adjacence d'un graphe écrite en langage Python et telle que si `M[i][j] = 1` alors il existe un arc qui va du sommet d'indice `i` au sommet d'indice `j`. Par exemple, `M[0][1] = 1` alors il existe un arc qui va du sommet d'indice `0` au sommet d'indice `1`.

```
M = [[0, 1, 0, 0, 0],
     [0, 0, 1, 0, 0],
     [0, 0, 0, 1, 0],
     [0, 1, 0, 0, 1],
     [0, 0, 0, 0, 0]]
```

6. Représenter le graphe associé à cette matrice d'adjacence. Les noms des sommets seront leurs indices.
7. Déterminer s'il est possible de trouver un ordre permettant de réaliser les tâches représentées par le graphe de la question 6 en respectant leurs dépendances. Si oui, donner l'ordre. Si non, expliquer pourquoi.

Voici le code Python d'une fonction `mystere`.

```
1  def mystere(graphe, s, n, ouverts, fermes, resultat):
2      """ Paramètres :
3          graphe un graphe représenté par une matrice d'adjacence
4          s l'indice d'un sommet du graphe
5          n le nombre de sommets du graphe
6          ouverts une liste de booléens permettant de savoir
7                  si le traitement d'un sommet a été commencé
8          fermes une liste de booléens permettant de savoir
9                 si le traitement d'un sommet a été terminé
10         Retour : False s'il y a eu un "problème", True sinon.
11         Le paramètre resultat sera modifié ultérieurement.
12     """
13     if ouverts[s]:
14         return False
15     if not fermes[s]:
16         ouverts[s] = True
17         for i in range(n):
18             if graphe[s][i] == 1:
19                 val = mystere(graphe, i, n, ouverts, fermes, resultat)
20                 if not val:
21                     return False
22         ouverts[s] = False
23         fermes[s] = True
24         # ...
25     return True
```

8. En utilisant la matrice `M` donnée précédemment, déterminer si la variable `ok` vaut `True` ou `False` à l'issue des instructions suivantes :

```
1 n = len(M)
2 ouverts = [False for i in range(n)]
3 fermes = [False for i in range(n)]
4 ok = mystere(M, 1, n, ouverts, fermes, None)
```

Décrire précisément les appels effectués à la fonction `mystere` et les valeurs des tableaux `ouverts` et `fermes` lors de chaque appel. On pourra recopier et compléter le tableau ci-dessous.

Appel `mystere`	variable `ouverts`	variable `fermes`
Avant l'appel `mystere`	`[F,F,F,F,F]`	`[F,F,F,F,F]`
`mystere(M,1,5,[F,F,F,F,F],[F,F,F,F,F],None)`	`[F,T,F,F,F]`	`[F,F,F,F,F]`
`mystere(M,2,5,[F,T,F,F,F],[F,F,F,F,F],None)`	`[F,T,T,F,F]`	`[F,F,F,F,F]`
...		

9. De manière générale, expliquer dans quel cas cette fonction `mystere` renvoie `False`.

L'objectif est d'utiliser la fonction `mystere` pour écrire une fonction `ordre_realisation` qui, lorsque c'est possible, détermine l'ordre de réalisation des tâches d'un graphe donné par sa matrice d'adjacence en respectant les dépendances entre les tâches.

Une structure de données de pile est représentée par une classe `Pile` qui possède les méthodes suivantes :

- la méthode `estVide` qui renvoie `True` si la pile représentée par l'objet est vide, `False` sinon ;
- la méthode `empiler` qui prend en paramètre un élément et l'ajoute au sommet de la pile ;
- la méthode `depiler` qui renvoie la valeur du sommet de la pile et enlève cet élément.

10. Déterminer la valeur associée à la variable `elt` après l'exécution des instructions suivantes :

```
>>> essai = Pile()
>>> essai.empiler(3)
>>> essai.empiler(2)
>>> essai.empiler(10)
>>> elt = essai.depiler()
>>> elt = essai.depiler()
```

Lorsqu'il en existe un, un ordre de réalisation des tâches sera représenté par un objet de classe `Pile` contenant tous les sommets du graphe de manière à ce que les tâches qu'il faut réaliser en premier se retrouvent au sommet de la pile.

La fonction `ordre_realisation` est écrite de la manière suivante :

```
1  def ordre_realisation(graphe):
2      n = len(graphe)
3      ouverts = [False for i in range(n)]
4      fermes = [False for i in range(n)]
5      ordre = Pile()
6      ok = True
7      s = 0
8      while (ok and s < n):
9          ok = mystere(graphe, s, n, ouverts, fermes, ordre)
10         s = s + 1
11     if ok :
12         return ordre
13     return None
```

11. Sachant que dans la fonction `mystere`, la ligne 24 peut être remplacée par une ou plusieurs instructions, donner ce qu'il faut écrire pour que, lorsque c'est possible, `ordre_realisation` renvoie effectivement un ordre de réalisation des tâches du graphe.

Exercice 3 (Programmation Python, POO, bases de données et SQL - 8 points)
Partie A
Une entreprise, présente sur différents sites en France, attribue à chacun de ses employés un numéro de badge unique.

Dans le tableau ci-dessous, on donne le numéro de badge, le nom, le prénom et les années de naissance et d'entrée dans l'entreprise de quelques salariés.

numéro badge	nom	prénom	année de naissance	année d'entrée
112	LESIEUR	Isabelle	1982	2005
2122	VASSEUR	Adrien	1962	1980
135	HADJI	Hakim	1992	2015

Pour chaque personne, on souhaite stocker les informations dans un objet de la classe `Personne` définie ci-dessous :

```
class Personne():
    def __init__(self, num, n , p , a_naiss, a_entree):
        self.num_badge = num
        self.nom = n
        self.prenom = p
        self.annee_naissance = a_naiss
        self.annee_entree = a_entree
```

1. Ecrire à l'aide du tableau précédent, l'instruction permettant de créer l'objet `personneA` de la première personne du tableau : LESIEUR Isabelle.
2. Donner l'instruction permettant d'obtenir le numéro de badge de l'objet `personneA` instancié à la question précédente.

On souhaite ajouter une méthode `annee_anciennete` à la classe `Personne` qui donne le nombre d'années d'ancienneté d'une personne au sein de l'entreprise. Par exemple : Madame LESIEUR Isabelle a une ancienneté dans l'entreprise de 19 ans en considérant que nous sommes en 2024.

3. Recopier et compléter le code suivant de la méthode `annee_anciennete` :

```
def annee_anciennete(self):
    return ...
```

On considère la classe `Personnel` qui modélise la liste du personnel d'une entreprise et dont le début de l'implémentation est la suivante :

```
class Personnel:
    def __init__(self):
        self.liste = []
```

4. Ecrire la méthode `ajouter` permettant d'ajouter un objet de type `Personne` à la liste du personnel de l'entreprise de la classe `Personnel`.
5. Ecrire la méthode `effectif` de la classe `Personnel`. Cette méthode devra renvoyer le nombre de personnes présentes dans l'entreprise.
6. Recopier et compléter la méthode `donne_nom` de la classe `Personnel`. Cette méthode prend en paramètre le numéro de badge d'une personne et renvoie le nom de la personne correspondant à ce badge si elle existe, ou `None` sinon.

```
def donne_nom(..., num):
    for elt in self.liste:
        if ... == num:
            return ...
    return ...
```

7. Lors de la célèbre cérémonie des voeux, l'entreprise souhaite mettre à l'honneur les personnes ayant exactement 10 ans d'ancienneté dans l'entreprise. Ecrire une méthode `nb_personne_honneur` de la classe `Personnel` qui prend en paramètre l'année de la cérémonie et qui retourne le nombre de personne(s) à mettre à l'honneur.
8. Ecrire une méthode `plus_anciens` de la classe `Personnel` qui retourne la liste des numéros de badge des personnes ayant la plus grande ancienneté dans l'entreprise.

Partie B

On utilise maintenant une base de données relationnelle. La table `Personnel` dont un extrait est donné ci-dessous contient toutes les données importantes sur le personnel de l'entreprise. L'attribut `num_centre` désigne le numéro du centre dans lequel travaille une personne.

Personnel					
num_badge	nom	prenom	num_centre	annee_naiss	annee_debut
112	LESIEUR	Isabelle	1	1982	2005
2122	VASSEUR	Adrien	2	1962	1980
135	HADJI	Hakim	1	1992	2015

L'attribut `num_badge` est la clé primaire pour la table `Personnel`.

9. Décrire par une phrase en français le résultat de la requête SQL suivante :

```
SELECT nom, prenom FROM Personnel WHERE num_centre = 2;
```

10. Monsieur HADJI Hakim vient d'obtenir une mutation pour le centre numéro 3. Donner la requête permettant de modifier son numéro de centre sachant que son numéro de badge est 135.

On souhaite proposer plus d'informations sur les différents centres de l'entreprise. Pour cela, on crée une seconde table `Centre` avec les attributs suivants :

- `num` de type INT ;
- `nom` de type TEXT ;
- `num_tel` de type TEXT ;
- `ville` de type TEXT.

Centre			
num	nom	num_tel	ville
1	Normandie	0450646859	Caen
2	PACA	0450646859	Marseille

11. Expliquer l'intérêt d'utiliser deux tables (`Personnel` et `Centre`) au lieu de regrouper toutes les informations dans une seule table.
12. Expliquer comment les tables `Centre` et `Personnel` sont mises en relation.
13. Ecrire une requête permettant d'avoir les noms des personnes travaillant dans le centre de Lille et ayant été embauchées entre 2015 (inclus) et 2020 (inclus).

Le centre de Normandie vient d'être fermé, mais les personnes de ce centre n'ont pas encore été affectées dans leur nouveau centre. On souhaite mettre à jour la table `Centre` en premier à l'aide de la requête suivante :

```
DELETE * FROM Centre WHERE nom = 'Normandie';
```

14. Expliquer pourquoi cette requête a renvoyé une erreur.