Centres étrangers - juin 2024 - sujet 2

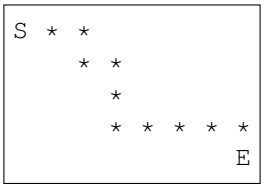
Exercice 1 (Programmation Python, POO et récursivité - 6 points)
On se déplace dans une grille rectangulaire. On s'intéresse aux chemins dont le départ est sur la case en haut à gauche et l'arrivée en bas à droite. Les seuls déplacements autorisés sont composés de déplacements élémentaires d'une case vers le bas ou d'une case vers la droite.

Un itinéraire est noté sous la forme d'une suite de lettres :

- `D` pour un déplacement vers la droite d'une case ;
- `B` pour un déplacement vers le bas d'une case.

Le nombre de caractères `D` est la longueur de l'itinéraire. Le nombre de caractères `B` est sa largeur.

Ainsi l'itinéraire `'DDBDBBDDDDB'` a pour longueur 7 et pour largeur 4. Sa représentation graphique est :



où :

- `S` représente la case de départ (start) ; ses coordonnées sont $(0;0)$;
- `*` représente les cases visitées ;
- `E` représente la case d'arrivée (end).

Partie A : programmation orientée objet
On représente un itinéraire avec la classe `Chemin` suivante :

```
class Chemin:

    def __init__(self, itineraire):
        self.itineraire = itineraire
        longueur, largeur = 0, 0
        for direction in self.itineraire:
            if direction == "D":
                longueur = longueur + 1
            if direction == "B":
                largeur = largeur +1
        self.longueur = longueur
        self.largeur = largeur
        self.grille = [['.' for i in range(longueur+1)] for j in range(largeur+1)]

    def remplir_grille(self):
        i, j = 0, 0 # Position initiale
        self.grille[0][0] = 'S' # Case de départ marquée d'un S
        for direction in ...:
            if direction == 'D':
                ... = ... # Déplacement vers la droite
            elif direction == 'B':
                ... = ... # Déplacement vers le bas
            self.grille[i][j] = '*' # Marquer le chemin avec '*'
        self.grille[self.largeur][self.longueur] = 'E' # Case d'arrivée marquée d'un E
```

1. Donner un attribut et une méthode de la classe `Chemin`.

On exécute le code ci-dessous dans la console Python :

```
chemin_1 = Chemin("DDBDBBDDDDB")
a = chemin_1.largueur
b = chemin_1.longueur
```

2. Préciser les valeurs contenues dans chacune des variables `a` et `b`.
3. Recopier et compléter la méthode `remplir_grille` qui remplace les `'.'` par des `'*'` pour signifier que le déplacement est passé par cette cellule du tableau.
4. Ecrire une méthode `get_dimensions` de la classe `Chemin` qui renvoie la longueur et la largeur de l'itinéraire sous la forme d'un tuple.
5. Ecrire une méthode `tracer_chemin` de la classe `Chemin` qui affiche une représentation graphique d'un itinéraire.

Partie B : génération aléatoire d'itinéraires

On souhaite créer des chemins de façon aléatoire. Pour cela, on utilise la méthode `choice` de la bibliothèque `random` dont on fournit ci-dessous la documentation.

```
`random.choice(sequence : list)`
Renvoie un élément choisi dans une liste non vide.
Si la population est vide, lève `IndexError`.
```

On rappelle que l'opérateur $\star$ permet de répéter une chaîne de caractères. Par exemple, on a :

```
>>> "Hello world ! " * 3
'Hello world ! Hello world ! Hello world ! '
```

L'algorithme proposé est le suivant :

- on initialise :
  - — une variable `itineraire` comme une chaîne de caractères vide,
  - — les variables `i` et `j` à `0` ;
- tant que l'on n'est pas sur la dernière ligne ou la dernière colonne du tableau :
  - — on tire au sort entre un déplacement à droite ou en bas,
  - — le déplacement est concaténé à la chaîne de caractères `itineraire`,
  - — si le déplacement est vers la droite, alors `j` est incrémenté de `1`,
  - — si le déplacement est vers le bas, alors `i` est incrémenté de `1` ;
- il reste à terminer le chemin en complétant par des déplacements afin d'atteindre la cellule en bas à droite.

6. Ecrire les lignes manquantes dans le code ci-dessous. Le nombre de lignes effacées dans le code n'est pas indicatif.

```
from random import choice

def itineraire_aleatoire(m, n):
    itineraire = ''
    i, j = 0, 0
    while i != m and j != n
        ... # il y a plusieurs lignes
    if i == m:
        itineraire = itineraire + 'D'*(n-j)
    if j == n:
        itineraire = itineraire + 'B'*(m-i)
    return itineraire
```

Partie C : calcul du nombre de chemins possibles

Soit m et n deux entiers naturels non nuls. On se place dans dans le contexte d'un itinéraire de longueur m et de largeur n, donc de dimension $m \times n$. On note $N(m,n)$ le nombre de chemins distincts respectant les contraintes de l'exercice.

7. Pour un itinéraire de dimension $1 \times n$ justifier, éventuellement à l'aide d'un exemple, qu'il y a un seul chemin, c'est-à-dire que, quel que soit n entier naturel, on a $N(1,n) = 1$.

De même, $N(m,1) = 1$.

8. Justifier que $N(m,n) = N(m-1,n) + N(m,n-1)$.
9. En utilisant les questions précédentes, écrire une fonction récursive `nombre_chemins(m, n)` qui renvoie le nombre de chemins possibles dans une grille rectangulaire de dimension $m \times n$.

Exercice 2 (POO, récursivité, arbres binaires et OS - 6 points)
Dans cet exercice, on travaille dans un environnement Linux. On considère l'arborescence de fichiers de la figure 1.

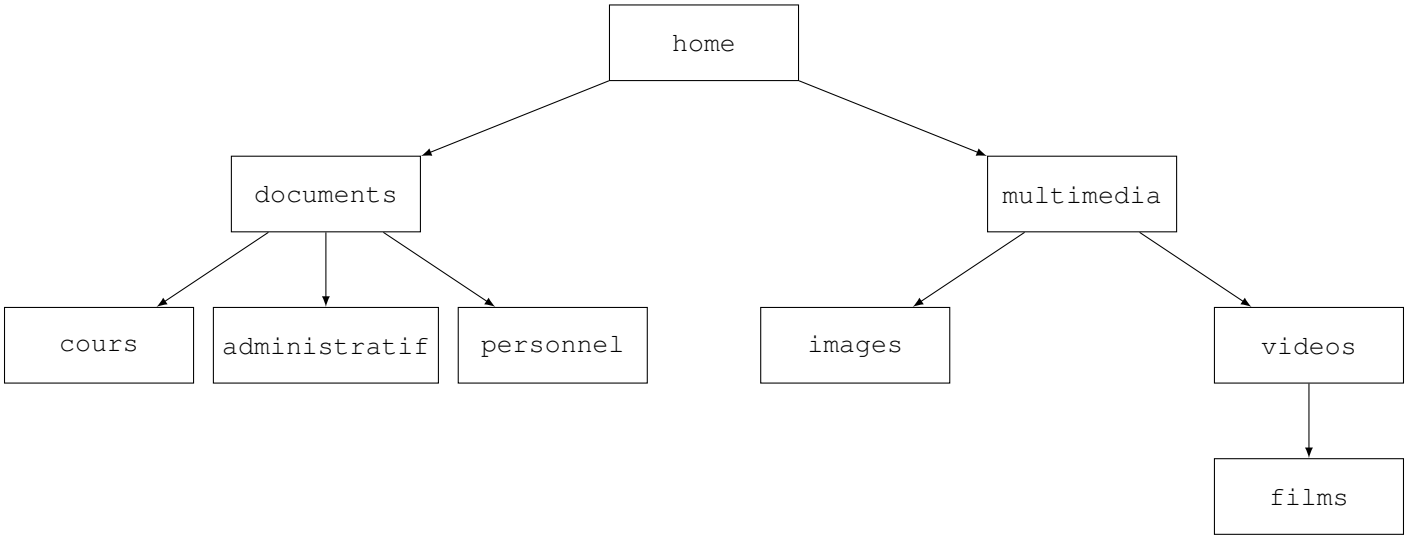


FIGURE 1 – Arborescence de fichiers

Partie A

1. Le répertoire courant est `home`. Donner une commande permettant de connaître le contenu du dossier `documents`.

On suppose que l'on se trouve dans le dossier `cours` et que l'on exécute la commande `mv ../../multimedia /home/documents`

2. Indiquer la modification que cela apporte dans l'arborescence de la figure 1.

On considère le code suivant :

```
1  class Arbre:
2      def __init__(self, nom, g, d):
3          self.nom = nom
4          self.gauche = g
5          self.droit = d
6
7      def est_vide(self):
8          return self.gauche is None and self.droite is None
9
10     def parcours(self):
11         print(self.nom)
12         if self.gauche != None:
13             self.gauche.parcours()
14         if self.droit != None:
15             self.droit.parcours()
```

3. Donner une raison qui justifie que le code précédent ne permet pas de modéliser l'arborescence de fichiers de la figure 1.
4. Donner le nom du parcours réalisé par le code précédent.
5. Donner la liste des dossiers dans l'ordre d'un parcours en largeur de l'arborescence. On ne demande pas d'écrire ce parcours en Python.

Partie B

Pour pouvoir modéliser l'arborescence de fichiers de la figure 1, on propose l'implémentation suivante. L'attribut `fils` est une variable de type `list` contenant tous les dossiers fils. Cette liste est vide dans le cas où le dossier est vide.

```
1  class Dossier:
2      def __init__(self, nom, liste):
3          self.nom = nom
4          self.fils = liste # liste d'objets de la classe Dossier
```

6. Ecrire le code Python d'une méthode `est_vide` qui renvoie `True` lorsque le dossier est vide et `False` sinon.
7. Ecrire le code Python permettant d'instancier une variable `var_multimedia` de la classe `Dossier` représentant le dossier `multimedia` de la figure 1. Attention : cela nécessite d'instancier tous les nœuds du sous-arbre de racine `multimedia`.
8. Recopier et compléter sur votre copie le code Python de la méthode `parcours` suivante qui affiche les noms de tous les descendants d'un dossier en utilisant l'ordre préfixe.

```
1 def parcours(self):
2     print(...)
3     for f in ...:
4         ...
```

9. Justifier que cette méthode `parcours` se termine toujours sur une arborescence de fichiers.
10. Proposer une modification de la méthode `parcours` pour que celle-ci effectue plutôt un parcours suffixe (ou postfixe).
11. Expliquer la différence de comportement entre un appel à la méthode `parcours` de la classe `Dossier` et une exécution de la commande UNIX `ls`.

On considère la variable `var_videos` de type `Dossier` représentant le dossier `videos` de la figure 1. On souhaite que le code Python `var_videos.mkdir('documentaires')` crée un dossier `documentaires` vide dans le dossier `var_videos`.

12. Ecrire le code Python de la méthode `mkdir`.
13. Ecrire en Python une méthode `contient(self, nom_dossier)` qui renvoie `True` si l'arborescence de racine `self` contient au moins un dossier de nom `nom_dossier` et `False` sinon.
14. Avec l'implémentation de la classe `Dossier` de cette partie, expliquer comment il serait possible de déterminer le dossier parent d'un dossier donné dans une arborescence donnée. On attend ici l'idée principale de l'algorithme décrite en français. On ne demande pas d'implémenter cet algorithme en Python.
15. Proposer une modification dans la méthode `__init__` de la classe `Dossier` qui permettrait de répondre à la question précédente beaucoup plus efficacement et expliquer votre choix.

Exercice 3 (Codage binaire, bases de données et SQL - 8 points)
Cet exercice est composé de deux parties peu dépendantes entre elles.

Lorsqu'il y a un accident, les pompiers essaient d'intervenir sur les lieux le plus rapidement possible avec les équipes et le matériel adéquats. On s'intéresse à l'étude d'un système informatique simplifié permettant de répondre à certaines de leurs problématiques.

Chaque pompier possède des aptitudes opérationnelles qui lui permettent de tenir un rôle. Lors du départ d'un véhicule (on parle d'agrès), il faut *a minima* un conducteur et un chef d'agrès.

Pour simplifier, on considère que

- un Véhicule Tout Usage (VTU) ne requiert que le duo conducteur et chef d'agrès, donc deux pompiers ;
- un Véhicule de Secours et d'Assistance aux Victimes (VSAV) requiert, en plus du duo conducteur et chef d'agrès, un équipier, donc trois pompiers ;
- un Fourgon Pompe Tonne (FPT) requiert, en plus du duo conducteur et chef d'agrès, un chef d'équipe et un équipier, donc quatre pompiers.

On souhaite entrer dans une base de données l'ensemble de ces informations afin de pouvoir conserver un historique des interventions.

Partie A : encodage binaire
Afin de gagner de la place mémoire, on décide de coder l'ensemble des aptitudes sur un entier de 8 bits plutôt que d'écrire en toutes lettres `équipier`, `chef d'équipe`, etc. Cet ensemble d'aptitudes formera la qualification du pompier considéré.

- Un personnel non formé est codé par 0 ;
- l'aptitude `équipier` est codée par le bit de rang 0 (celui de poids le plus faible), soit 2^0 ;
- l'aptitude `chef d'équipe` est codée par le bit de rang 1, soit 2^1 ;
- l'aptitude `chef d'agrès` est codée par le bit de rang 2, soit 2^2 ;
- enfin, l'aptitude `conducteur` est codée par le bit de rang 3, soit 2^3.

Remarque : un chef d'équipe étant nécessairement équipier, on lui ajoute cette aptitude dans son codage. De même, un chef d'agrès est nécessairement un chef d'équipe et un équipier : on lui ajoute ces aptitudes dans son codage.

1. Justifier que la qualification décimale 11 correspond à un chef d'équipe conducteur.
2. Déterminer le codage décimal de la qualification chef d'agrès conducteur.
3. Expliquer pourquoi dans la situation décrite, un pompier ne peut pas avoir de qualification dont le codage décimale est 4.
4. Avec ce codage sur un octet, indiquer combien de nouvelles aptitudes peuvent être définies.

On considère que chacune des quatre aptitudes aurait pu être codée par une chaîne de 10 caractères dont chaque caractère utilise un octet en mémoire.

5. Choisir, avec justification, l'économie mémoire que le codage sur un entier de 8 bits permet de faire comparée au codage basé sur les chaînes de caractères : environ 10%, 30%, 50% ou 98% ?

Partie B
On considère maintenant la base de données relationnelle suivante :

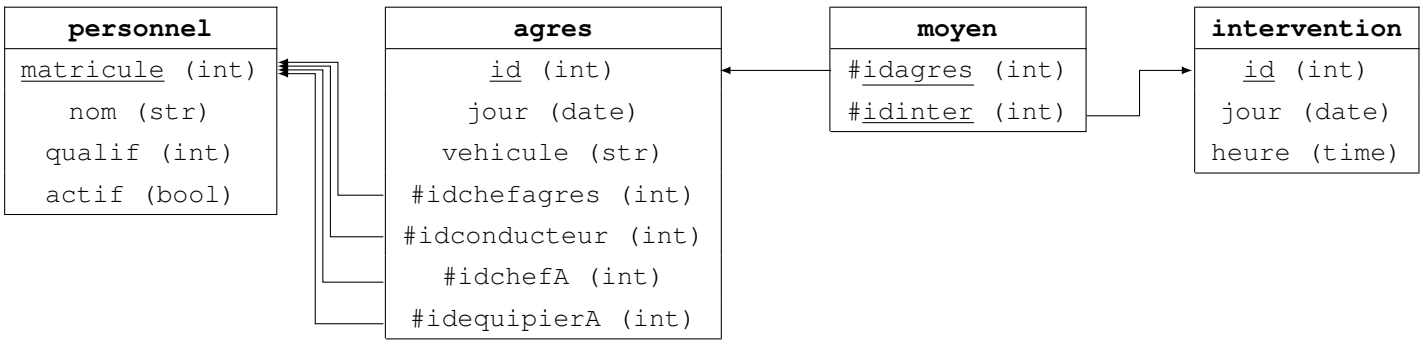


FIGURE 1 – Schéma relationnel

La table `personnel` est composée :

- du matricule unique du pompier ;
- de son nom ;
- de sa qualification (selon le codage vu avant) ;
- d'un attribut qui précise s'il est actif (1) ou inactif (0).

La table `agres` est composée :

- de son identifiant ;
- du jour où l'agrès se tient prêt à intervenir ;
- du type de véhicule ;
- de l'identifiant du chef d'agrès ;
- de l'identifiant du conducteur ;
- de l'identifiant du chef d'équipe si le véhicule le nécessite, sinon le champ est à NULL ;
- de l'identifiant de l'équipier si le véhicule le nécessite, sinon le champ est à NULL.

La table `moyen` est composée :

- de l'identifiant de l'agrès appelé sur intervention ;
- de l'identifiant de l'intervention.

La table `intervention` est composée :

- de son identifiant ;
- du jour de début d'intervention (sauf pour les longues interventions où il correspond au jour de l'agrès) ;
- de l'heure de début d'intervention.

On rappelle que `COUNT(*)` permet de compter le nombre de lignes extraites lors d'une requête. Par exemple, pour afficher le nombre de personnes dans la table `personnel`, on exécute la requête :

```
SELECT COUNT(*) FROM personnel;
```

On rappelle également que `DISTINCT` permet de retirer les doublons des réponses. Par exemple, pour afficher tous les noms distincts de la table `personnel`, on exécute la requête :

```
SELECT DISTINCT(nom) FROM personnel;
```

On considère l'extrait de la base de données ci-dessous :

personnel			
matricule	nom	qualif	actif
10	Sam	3	1
16	Charlot	1	0
31	Red	23	0
83	Vaillante	7	1
2501	Marco	1	1
2674	Aicha	23	1
3004	Fatou	7	1
4044	Abdel	19	1
4671	Mamadou	17	1
5301	Zoe	17	1
7450	Medhi	3	1
8641	Gaia	1	1
8678	Kevin	17	0
8682	Marie	1	1
9153	Fred	23	1

moyen	
idagres	idinter
0	0
2	3
2	4
3	4
4	4
9	5
17	6
22	7
23	8
24	8
24	9

intervention		
id	jour	heure
0	2023-11-21	12:32:21
1	2023-11-22	22:20:00
2	2023-12-17	23:17:30
3	2024-02-15	01:44:06
4	2024-02-15	12:15:00
5	2024-03-02	04:58:12
6	2024-03-27	13:07:18
7	2024-05-31	05:17:12
8	2024-06-11	05:38:17
9	2024-06-11	15:08:56
10	2024-06-18	07:42:33

agres						
id	jour	vehicule	idchefagres	idconducteur	idchefA	idequipierA
0	2023-11-21	VSAV	83	9153	NULL	10
2	2024-02-15	VSAV	2674	4044	NULL	8641
3	2024-02-15	FPT	9153	5301	8682	2501
4	2024-02-15	VSAV	83	4671	NULL	7450
7	2024-02-29	VSAV	9153	3004	NULL	2501
9	2024-03-02	FPT	2674	5301	8682	8641
12	2024-03-21	VTU	3004	5301	NULL	NULL
17	2024-03-27	VSAV	3004	8682	NULL	10
18	2024-03-27	VSAV	9153	5301	NULL	10
22	2024-05-31	FPT	9153	4044	7450	8641
23	2024-06-11	VTU	83	2674	NULL	NULL
24	2024-06-11	VSAV	3004	4044	NULL	7450

6. Expliquer la différence entre une clé primaire et une clé étrangère.
7. Expliquer pourquoi la requête suivante génère une erreur pour l'extrait de données :

```
INSERT INTO moyen(idagres, idinter) VALUES (1,5);
```

8. Proposer une requête SQL qui met à jour l'heure de l'intervention du 15 février 2024 de 01 heure 44 minutes et 06 secondes à 10 heures 44 minutes et 06 secondes.
9. Préciser le résultat de la requête suivante pour l'extrait de données :

```
SELECT nom FROM personnel WHERE actif = 0;
```

10. Proposer une requête SQL qui permet d'afficher les noms des personnels conducteurs actifs. On notera qu'un conducteur possède un attribut `qualif` supérieur ou égal à 16.
11. Ecrire l'affichage obtenu après exécution des deux requêtes ci-dessous sur l'extrait de la base de données. Expliquer ce que chacune des requêtes affiche en général.
    - ⋆ **Requête A :**

```
SELECT COUNT(*) FROM agres
WHERE jour = '2024-03-27';
```

    - ⋆ **Requête B :**

```
SELECT COUNT(*) FROM moyen AS m
INNER JOIN agres AS a ON a.id = m.idagres
WHERE a.jour = '2024-03-27';
```

12. Proposer une requête SQL qui renvoie sans répétition tous les noms des chefs d'agrès assignés à un véhicule le 15 février 2024.
13. Proposer une requête SQL qui renvoie sans répétition tous les noms des chefs d'agrès engagés en intervention le 11 juin 2024.