BACCALAURÉAT GÉNÉRAL

ÉPREUVE D'ENSEIGNEMENT DE SPÉCIALITÉ

SESSION 2026

NUMÉRIQUE ET SCIENCES INFORMATIQUES

ÉPREUVE DU MERECREDI 09 SEPTEMBRE 2026

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 17 pages numérotées de 1/17 à 17/17.

Le sujet est composé de trois exercices indépendants.

Le candidat traite les trois exercices.

Exercice 1 (6 points)

Cet exercice porte sur les bases de données, les graphes, les files et la programmation orientée objet.

Dans cet exercice, 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`) et `JOIN ... ON` ;
- construire des requêtes d'insertion et de mise à jour à l'aide de `UPDATE`, `INSERT` et `DELETE` ;
- affiner les recherches à l'aide de `DISTINCT` et `ORDER BY`.

Une entreprise spécialisée dans les solutions d'évacuation d'urgence et de sécurité incendie possède trois sites situés à Paris, Bordeaux et Lyon. Elle gère des plans d'évacuation pour chaque bâtiment. Dans la partie A, on s'intéresse à une base de données permettant de gérer le stockage de ces plans ; dans la partie B, on s'intéresse à la modélisation de chaque plan d'évacuation sous la forme de graphe.

Partie A : Base de données

Le schéma relationnel de la base de données utilisé par l'entreprise est représenté figure 1. Les clés primaires sont soulignées et les éventuelles clés étrangères sont précédées du symbole "#".

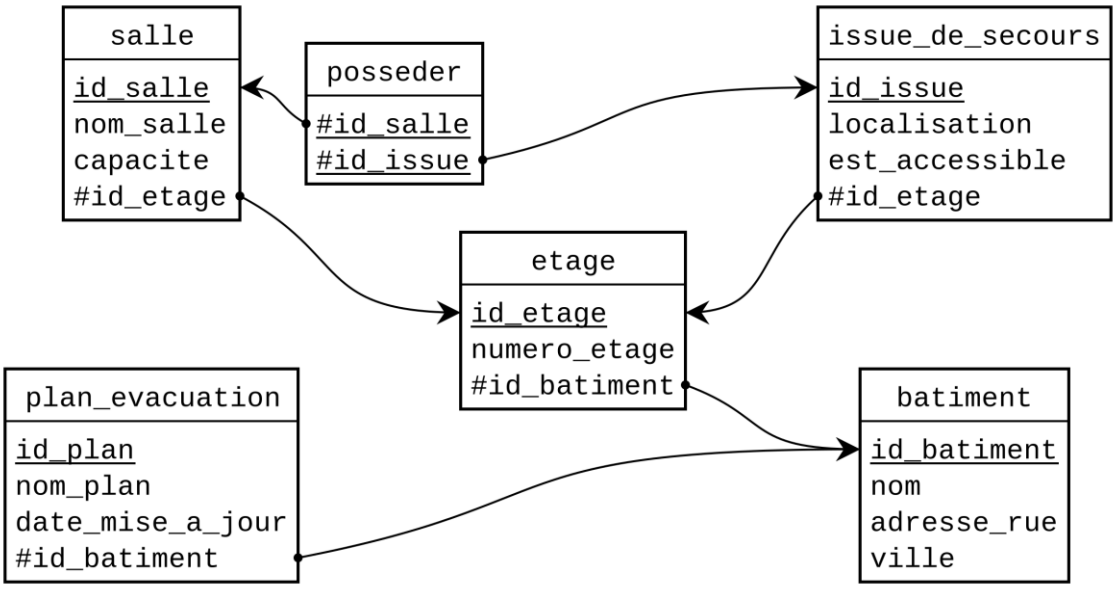

Figure 1. Schéma de la base de données

Les relations de la base de données sont indiquées par des flèches et permettent de prendre en compte les éléments suivants :

- un bâtiment peut avoir plusieurs étages ;

- un étage peut contenir plusieurs salles et issues de secours ;
- une salle peut posséder une ou plusieurs issues de secours ;
- chaque plan d'évacuation est associé à un bâtiment.

Extrait de données

Table `batiment`

id_batiment	nom	adresse_rue	ville
1	Bâtiment A	123 rue principale	Paris
2	Bâtiment B	45 rue principale	Paris
3	Bâtiment C	12 rue du bouchon	Lyon

Table `etage`

id_etage	numero_etage	id_batiment
1	1	1
2	2	1
3	3	1
4	1	2

Table `salle`

id_salle	nom_salle	capacite	id_etage
1	Salle 101	30	3
2	Salle 102	25	3
3	Salle conférence	100	2

Table `issue_de_secours`

id_issue	localisation	est_accessible	id_etage
1	Nord	TRUE	3
2	Sud	TRUE	3
3	Est	FALSE	2

Table `plan_evacuation`

id_plan	nom_plan	date_mise_a_jour	id_batiment
1	Plan général bâtiment A	2025-06-01	1

Table `posseder`

id_salle	id_issue
1	1
1	2
2	1
3	3

1. Donner la clé primaire et les éventuelles clés étrangères de la relation `salle`.
2. Citer une contrainte de domaine, une contrainte de relation et une contrainte de référence que doivent respecter les données de la relation `salle`.
3. Écrire une requête SQL permettant d'afficher l'adresse de chaque bâtiment situé à Paris.
4. Recopier et compléter la requête suivante permettant d'afficher le nom des salles, leur numéro étage et le nom du bâtiment, mais uniquement pour les bâtiments situés à Paris.

```
SELECT ...
FROM salle
JOIN ...
JOIN ...
WHERE ville= ...
```

5. Écrire une requête SQL qui affiche le nom du bâtiment associé au plan d'évacuation nommé "Plan général bâtiment A".
6. Écrire une requête SQL permettant de lister les noms des salles et la localisation des issues de secours associées.
7. Écrire une requête SQL permettant d'afficher la localisation des issues de secours accessibles pour l'étage vérifiant `id_etage=2`.

Partie B : Graphes et parcours d'évacuation

Dans cette partie, on cherche à construire le plan d'évacuation d'un bâtiment possédant une seule sortie de secours. Pour simplifier l'analyse, on modélise les salles et les couloirs par un graphe :

Les deux variables `batiment` et `sortie` ci-après représentent le bâtiment et la sortie.

```
 1 batiment = {
 2     'A': ['B', 'C'],
 3     'B': ['A', 'D', 'E'],
 4     'C': ['A', 'E'],
 5     'D': ['B', 'E', 'F'],
 6     'E': ['B', 'C', 'D', 'G'],
 7     'F': ['D'],
 8     'G': ['E']
 9 }
10 sortie = 'F'
```

8. Représenter graphiquement un graphe correspondant à la variable `batiment`.

On rappelle que, dans un graphe, la distance entre deux sommets `A` et `B` est le nombre minimal d'arêtes empruntées pour aller du sommet `A` au sommet `B`.

9. Déterminer, pour chaque salle, le plus court chemin vers la sortie `F` et calculer la distance correspondante.

Une fonction `calculer_distances` détermine les distances entre un sommet et les autres sommets d'un graphe. On donne ci-dessous un exemple d'appel dans la console.

```
>>> calculer_distances(batiment, 'F')
{'F': 0, 'D': 1, 'B': 2, 'E': 2, 'A': 3, 'C': 3, 'G': 3}
```

On donne une partie du code de la fonction `calculer_distances`. Cette fonction utilise une classe `File`, dont on rappelle les méthodes de base :

- `enfiler(elt)`
- `defiler()`
- `est_vide()`

```
 1 def calculer_distances(graphe, sortie):
 2     """Calcule les distances
 3     entre la sortie et tous les autres sommets"""
 4     distances = {sortie: 0}
 5     file = File()
 6     file.enfiler(...)
 7
 8     while not file.est_vide():
 9         s = file.defiler()
10         for v in graphe[s]:
11             if v not in distances:
12                 distances[v] = ...
13                 file.enfiler(...)
14
15     return distances
```

10. Recopier sur la copie et compléter les lignes 6, 12 et 13 de la fonction `calculer_distances`.

Exercice 2 (6 points)

Cet exercice porte sur la sécurisation des communications, la programmation orientée objet et les dictionnaires en Python.

Les deux parties sont indépendantes.

Partie A

Le principe du *chiffrement XOR* consiste à chiffrer chaque caractère du message en le combinant, bit à bit, avec la clé à l'aide de l'opérateur logique « ou exclusif ». La clé est un caractère ASCII dont la représentation binaire sur un octet est utilisée pour effectuer cette opération sur chacun des caractères du message.

On rappelle la table de vérité de l'opérateur logique « ou exclusif », noté XOR, où A et B sont deux variables logiques pouvant prendre les valeurs `0` et `1`:

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Dans cette partie, on se limite aux caractères ASCII, chacun étant représenté sur un octet.

Exemple :

On considère la lettre `M` dont la représentation binaire est `01001101` et la clé `00110100` qui représente le caractère `4`.

```
              M : 0 1 0 0 1 1 0 1
              4 : 0 0 1 1 0 1 0 0
message chiffré : 0 1 1 1 1 0 0 1
```

Le message est chiffré par l'octet `01111001` qui correspond au caractère ASCII `y`. `M` sera donc chiffré par `y`.

1. Déterminer la représentation binaire sur un octet du caractère chiffré obtenu en appliquant le chiffrement XOR au message `T` par la clé `N`. On rappelle que le code ASCII de la lettre `T` est `01010100` et que celui de la lettre `N` est `01001110`.

2. Recopier et compléter la dernière colonne de la table de vérité suivante ` :

A	B	A XOR B	(A XOR B) XOR B

A	B	A XOR B	(A XOR B) XOR B
0	0	0	
0	1	1	
1	0	1	
1	1	0	

3. Déduire de la question précédente que cet algorithme de chiffrement est symétrique.

Partie B

Dans cette partie, on se limite aux lettres majuscules de l'alphabet latin utilisées en français.

Le *carré de Polybe* est une méthode de chiffrement qui remplace chaque lettre de l'alphabet par une paire de chiffres correspondant à sa position dans une grille 5x5, d'abord la ligne puis la colonne. Les lettres `I` et `J` sont regroupées pour avoir exactement 25 cases et sont donc confondues.

Le carré de Polybe initial est donné ci-dessous.

	1	2	3	4	5
1	A	B	C	D	E
2	F	G	H	I/J	K
3	L	M	N	O	P
4	Q	R	S	T	U
5	V	W	X	Y	Z

Afin d'éviter d'utiliser toujours la même grille, ce qui rendrait la méthode trop prévisible et donc vulnérable, on personnalise le carré de Polybe à l'aide d'un mot servant de clé. Pour cela, on inscrit dans la grille les lettres de ce mot, en respectant leur ordre d'apparition et sans répétition, puis on complète avec les lettres restantes de l'alphabet dans l'ordre alphabétique.

On fait le choix dans cet exercice de ne pas traiter la lettre `J`. Les messages à chiffrer ne contiendront donc pas de `J`.

Par exemple, avec la clé `SECRET`, on obtient la grille suivante :

	1	2	3	4	5
1	S	E	C	R	T
2	A	B	D	F	G
3	H	I	K	L	M
4	N	O	P	Q	U
5	V	W	X	Y	Z

Ainsi, pour chiffrer le message `LW`, on chiffre chacune des lettres le composant :

- `L` est en position ligne 3, colonne 4. `L` est donc chiffré par 34 ;
- `W` est en position ligne 5, colonne 2. `W` est donc chiffré par 52.

Le message chiffré est donc 3452.

4. Chiffrer le mot `NSI` à l'aide du carré de Polybe construit avec la clé `SECRET`.

5. Retrouver le message en clair correspondant au message chiffré 222113 à l'aide du carré de Polybe construit avec la clé `SECRET`.

On va représenter les grilles à l'aide de la classe suivante dont les méthodes seront explicitées au fur et à mesure de l'exercice.

```
class Grille:
    def __init__(self, cle):
        self.cle = cle
        self.lettres = []
        self.generer_lettres()
        self.l_vers_c, self.c_vers_l = {}, {}
        self.generer_dicos()
```

Cette classe possède une méthode `chiffrer` qui prend en paramètre `self` et une chaîne de caractères `message_clair` correspondant au message à chiffrer, et renvoie une chaîne de caractères correspondant au message chiffré.

6. Dans cette question on considère la classe `Grille` entièrement implémentée.

   Écrire les instructions Python permettant alors :

   - de créer une instance `grille` de la classe `Grille` avec la clé `SECRET` ;
   - d'utiliser la méthode `chiffrer` pour chiffrer le message `NSI` et stocker le résultat dans la variable `message_chiffre`.

Cet instance `grille` sera utilisée pour les exemples dans la suite de cet exercice.

La construction de la grille repose sur la création d'une liste, enregistrée dans l'attribut `lettres`. Celle-ci contient, dans l'ordre, les lettres distinctes de la clé (dans leur ordre d'apparition) suivies des lettres restantes de l'alphabet, à l'exception de `J`. La méthode `generer_lettres` permet de générer cette liste.

7. Recopier et compléter les lignes 6 et 7 de la méthode `generer_lettres`.

```
1     def generer_lettres(self):
2         for c in self.cle:
3             if c not in self.lettres:
4                 self.lettres.append(c)
5         for c in "ABCDEFGHIKLMNOPQRSTUVWXYZ":
6             if ...:
7                 ...
```

Pour chiffrer et déchiffrer des messages, on utilise deux dictionnaires `l_vers_c` et `c_vers_l` qui servent respectivement à associer une lettre à des coordonnées et des coordonnées à une lettre. Ces deux dictionnaires sont des attributs des objets de la classe `Grille` et sont construits à l'aide de la méthode `generer_dicos`. Les coordonnées sont exprimées à l'aide d'une chaîne de caractères.

Exemples :

- `grille.l_vers_c['S']` vaut `'11'` (ligne 1, colonne 1)
- `grille.c_vers_l['34']` vaut `'L'` (ligne 3, colonne 4)

8. Recopier et compléter les lignes 4, 6, 7 et 8 de la méthode `generer_dicos`.

```
1     def generer_dicos(self):
2         for i in range(len(self.lettres)):
3             ligne = i//5 + 1
4             col = ... + 1
5             lettre = self.lettres[i]
6             coord = str(ligne) + ...
7             self.l_vers_c[lettre] = ...
8             self.c_vers_l[coord] = ...
```

Pour rappel, la méthode `chiffrer` qui prend en paramètre `self` et une chaîne de caractères `message_clair` correspondant au message à chiffrer, et renvoie une chaîne de caractères correspondant au message chiffré. Si `message_clair` contient des caractères autre que des lettres de la grille, ils sont ignorés lors du chiffrement. Dans l'exemple ci-dessous, l'espace est ignoré.

```
>>> grille.chiffrer("UN EXEMPLE")
'454112531235433412'
```

9. Écrire le code de la méthode `chiffrer`.

10. Recopier et compléter les lignes 4, 5, 6, 8 et 9 de la méthode `dechiffrer` ci-dessous. Cette fonction déchiffre le message chiffré passé en paramètre.

```
1 def dechiffrer(self, message_chiffre):
2     message_clair = ""
3     for i in range(0, len(message_chiffre), 2):
4         coord = message_chiffre[i] + ...
5         if ...:
6             message_clair += ...
7         else:
8             message_clair += ...
9     return ...
```

Rappel : dans la boucle

```
for i in range(0, len(message_chiffre), 2):
    ...
```

la variable `i` prend successivement les valeurs 0, 2, 4, 6, … c'est-à-dire avec un pas de 2.

Exercice 3 (8 points)

Cet exercice porte sur les structures de données, la programmation de base en Python, l'algorithmique et les commandes Unix.

Partie A

Une couleur à l'écran est fabriquée en mélangeant trois lumières de base : Red (rouge), Green (vert) et Blue (bleu). Chaque composante est codée par un entier entre 0 (pas de lumière) et 255 (intensité maximale).

En combinant ces trois nombres, on peut représenter n'importe quelle couleur.

Par exemple :
* La couleur noire est codée par `(0, 0, 0)` ;
* la couleur blanche par `(255, 255, 255)` ;
* la couleur grise par `(128, 128, 128)` ;
* la couleur jaune par `(255, 255, 0)`.

À chaque image, on associe la liste des triplets RGB de tous ses pixels ligne par ligne (les pixels de la première ligne, puis de la deuxième ligne, …).

Par exemple, l'image ci-dessous est une image de 2 pixels par 2 pixels contenant uniquement les couleurs : blanc, gris et noir.

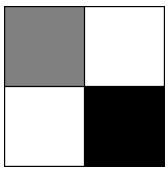

Figure 1. carre_2x2.png

La liste associée à cette image est :

```
carre = [(128, 128, 128),(255, 255, 255), (255, 255, 255), (0, 0, 0)]
```

On considère l'image ci-dessous :

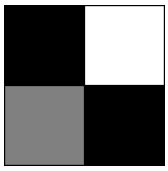

Figure 2. carre_2x2_bis.png

1. Donner la liste `carre_bis` associée à l'image `carre_2x2_bis.png`.
2. Donner l'instruction (à saisir dans la console) permettant d'accéder à la composante rouge de la case grise de l'image `carre_2x2_bis.png`.

On importe le module `Image` de la bibliothèque `PIL` (Pillow) dont on utilise :

- la fonction `Image.open` qui prend en paramètre le nom d'un fichier image (ou un chemin vers ce fichier) et qui renvoie un objet de type `PIL.Image.Image`.
- la méthode `getdata()` s'applique à un objet de type `PIL.Image.Image` et renvoie un objet de type `ImagingCore` contenant tous les pixels de l'image, rangés ligne par ligne. Pour pouvoir manipuler ces pixels comme une vraie liste Python, il faut convertir cet objet avec `list()`.

La fonction `convertir` prend en paramètre le nom d'un fichier image (type `str`) et elle renvoie la liste des codes RGB de chaque pixel de cette image.
Par exemple :

```
>>> convertir("carre_2x2.png")
[(128, 128, 128),(255, 255, 255), (255, 255, 255), (0, 0, 0)]
```

3. Recopier et compléter les lignes 4 et 6 de la fonction `convertir` :

```
1 from PIL import Image
2 def convertir(fichier_image):
3     # ouverture de l'image
4     image = ...
5     # création liste des codes RGB
6     l_pixels = list(image.getdata())
7     return ...
```

Comme chaque pixel d'une image est codé en RGB, on parcourt l'image pixel par pixel, à chaque pixel on accède à la valeur du rouge. On calcule ensuite la moyenne de ces valeurs. On obtient alors la moyenne du rouge pour l'image.

Par exemple, pour l'image `carre_2x2.png`, la moyenne du rouge est 159.5.

4. Écrire le calcul permettant d'obtenir la moyenne du rouge pour l'image `carre_2x2_bis.png`.

La fonction `moyenne_rouge` prend en paramètre le nom d'une image (type `str`) et elle renvoie la moyenne du rouge pour cette image.

5. Écrire la fin de la fonction `moyenne_rouge` :

```
1 def moyenne_rouge(fichier_image):
2     pixels = convertir(fichier_image)
3     ... # plusieurs lignes à ajouter
4     return ...
```

Sur le même principe, on crée les fonctions `moyenne_vert` et `moyenne_bleu` que l'on ne demande pas de coder.

6. Recopier la ligne 2 de la fonction `moyenne_RGB` qui prend en paramètre le nom d'une image (type `str`) et qui renvoie un tuple contenant, dans cet ordre, la moyenne du rouge, du vert et du bleu des pixels de cette image.

```
1 def moyenne_RGB(fichier_image) :
2     return ...
```

On considère l'arborescence du système de fichiers de l'ordinateur personnel de Paul ci-dessous.

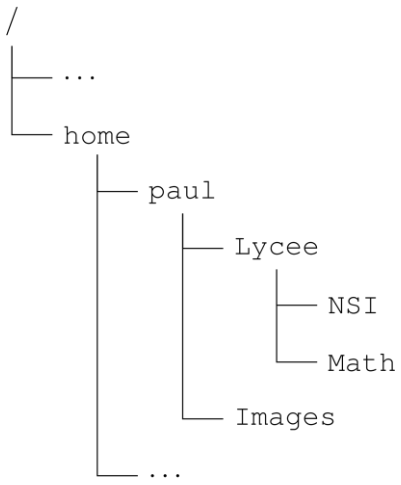

Figure 3. Arborescence de l'ordinateur de Paul

Sur son ordinateur, Paul a codé les fonctions précédentes dans un fichier `moy.py` qu'il a enregistré dans son dossier `NSI`. Il souhaite maintenant les tester sur sa photo de vacances `mer.jpg`, qui se trouve dans son dossier `Images`.
Depuis le dossier `NSI`, il ouvre un interpréteur Python et saisit l'instruction suivante :

```
>>> from moy import moyenne_RGB
```

7. Écrire la commande qu'il doit saisir dans l'interpréteur pour obtenir la moyenne RGB de sa photo `mer.jpg`.

Finalement, Paul décide de déplacer sa photo `mer.jpg` dans son dossier `NSI`.
Il ouvre un terminal et se rend dans le dossier `paul`.

8. Écrire la commande UNIX qu'il doit saisir dans un terminal pour réaliser ce déplacement.

Partie B

Une photo numérique est composée de plusieurs millions de pixels, ainsi les fonctions de la partie précédente pourront être utilisées.

Le but de cette partie est d'utiliser l'algorithme des **k-plus proches voisins** (k-PPV) pour prédire si une photo représente une pomme verte, une cerise ou une orange grâce à sa couleur dominante.

On crée un jeu d'entraînement avec de nombreuses photos de fruits dont on a déterminé la catégorie (dans notre exemple nous avons trois catégories de fruits : `pomme_verte, cerise ou orange`). Ce jeu d'entraînement est implémenté par un dictionnaire dont la clé est le nom du fichier de la photo et la valeur associée est une liste contenant dans cet ordre :

- La catégorie de la photo
- la liste des moyennes RGB obtenue avec la fonction `moyenne_RGB`

Voici un extrait du jeu d'entraînement qui sera utilisé :

```
jeu_entrainement = {
    'image1.jpg' : ['pomme_verte', moyenne_RGB('image1.jpg')],
    'image2.jpg' : ['cerise', moyenne_RGB('image2.jpg')],
    'image3.jpg' : ['orange', moyenne_RGB('image3.jpg')],
    ...
    }
```

9. Donner la commande (à saisir dans la console) permettant d'obtenir la liste des moyennes RGB de `image1.jpg`.

On considère deux listes `l1` et `l2` contenant des moyennes RGB :

```
l1 = [r1, g1, b1]
l2 = [r2, g2, b2]
```

Pour calculer la distance euclidienne entre `l1` et `l2`, on utilise la formule suivante :
$d = \sqrt{(r2 - r1)^2 + (g2 - g1)^2 + (b2 - b1)^2}$

Pour utiliser la fonction racine carrée (sqrt), on l'importe de la façon suivante : `from math import sqrt`

10. Écrire une fonction `distance` qui prend en paramètre deux tuples t`1` et t`2` contenant les valeurs moyennes RGB de deux images et qui renvoie la distance euclidienne entre ces deux tuples.
11. Écrire la commande permettant de calculer la distance entre les photos `image1.jpg` et `image2.jpg` stockées dans le dictionnaire `jeu_entrainement`.

On considère une photo `nouvelle_photo.jpg` dont on ne connait pas la catégorie. On crée une liste, nommée `l_distance`, contenant des tuples de la forme `(distance, nom_fichier)` où :

- `nom_fichier` est le nom des photos de `jeu_entrainement`
- `distance` est la distance entre la nouvelle photo et celle de nom_fichier

Par exemple, pour la photo `clementine.jpg`, on obtient la liste suivante :

```
l_distance = [(41.158, 'image1.jpg'), (27.166, 'image2.jpg'),
(30.43, 'image3.jpg'), (54.791, 'image4.jpg'), (26.87,
'image5.jpg'), (41.158, 'image6.jpg')]
```

La fonction `creation_liste_distance` prend en paramètres le dictionnaire `jeu_entrainement` et le nom du fichier d'une photo `nouvelle_photo` (type str) et elle renvoie la liste `l_distance` comme définie précédemment.

12. Recopier et compléter les lignes 2, 4, 5 et 6 de la fonction `creation_liste_distance`

```
1 def creation_liste_distance(jeu_entrainement,
nouvelle_photo):
2     l_RGB = moyenne_RGB(...)
3     l_distance = []
4     for nom_fichier in ... :
5         d = distance(...)
6         l_distance.append(...)
7     return l_distance
```

`sort()` est une méthode qui modifie la liste en la triant par ordre croissant. On admet que la liste les tuples (ici des 2-uplets) est trié selon la première valeur du tuple. Cette méthode pourra être utilisée dans les questions qui suivent.

La fonction `kppv` prend en paramètres un entier naturel `k` non nul, le dictionnaire `jeu_entrainement`, le nom du fichier d'une photo `nouvelle_photo` (type `str`) et renvoie la liste `l_kppv` contenant les tuples `(distance, nom_fichier)` des k-plus proches voisins de cette nouvelle photo. Par exemple :

```
>>> l_kppv = kppv(3, jeu_entrainement, "clementine.jpg")
[(11.489, 'image9.jpg'), (26.87, 'image5.jpg'), (27.166,
'image2.jpg')]
```

13. Écrire la fonction `kppv`.

Une fois les k-plus proches voisins récupérés, il faut prédire la catégorie de la nouvelle image : `pomme_verte`, `cerise` ou `orange`.

14. Écrire une fonction `maximum_dico` qui prend en paramètre un dictionnaire `d` dont les clés sont des noms de fruit (de type `str`) et les valeurs sont des nombres entiers naturels. Cette fonction renvoie la liste des noms de fruits ayant la plus grande valeur associée.

Par exemple :

```
>>> d
{'pomme_verte': 1, 'cerise': 0, 'orange': 1}
>>> maximum_dico(d)
['pomme_verte', 'orange']
```

La fonction `predire` prend en paramètre le dictionnaire `jeu_entrainement` et une liste `l_kppv` contenant les noms de fichiers des k plus proches voisins de cette nouvelle photo. Cette fonction renvoie la liste contenant le ou les noms des catégories probables pour cette nouvelle photo en fonction du jeu d'entraînement utilisé. Par exemple,

```
>>> predire(jeu_entrainement, l_kppv)
['orange']
```

15. Recopier et compléter les lignes 3 à 7 de la fonction `predire`.

```
1 def predire(jeu_entrainement, l_kppv):
2     prediction = {'pomme_verte' : 0, 'cerise' : 0,
'orange' : 0}
3     for couple in l_kppv :
4         nom_fichier = ...
5         fruit = ..
6         prediction[fruit] += 1
7     return ...
```

On considère la photo `mystere.jpg` ci-dessous où l'on voit des cerises et des feuilles de cerisier.



Figure 4. mystere.jpg

On exécute les lignes de commandes ci-dessous :

```
>>> L_kppv = kppv(3, jeu_entrainement, "image11.jpg")
>>> predire(jeu_entrainement, L_kppv)
['pomme']
```

16. Expliquer ce résultat et donner une idée pour améliorer cet algorithme.