BACCALAURÉAT GÉNÉRAL

ÉPREUVE D'ENSEIGNEMENT DE SPÉCIALITÉ

SESSION 2025

NUMÉRIQUE ET SCIENCES INFORMATIQUES

JOUR 2

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 21 pages numérotées de 1/21 à 21/21.

Le sujet est composé de trois exercices indépendants.

Le candidat traite les trois exercices.

Exercice 1 (6 points)

Cet exercice porte sur les tableaux, les dictionnaires, les arbres binaires, la programmation en Python et la récursivité.

Lors de la transmission de données, des erreurs peuvent se glisser.

On se propose d'étudier des techniques permettant de minimiser les conséquences de telles erreurs.

Partie A

Pour encoder un texte en binaire, on traduit chaque caractère en un octet, par exemple en utilisant le code ASCII. La table ASCII permet de traduire les caractères classiques en entiers compris entre 0 et 127, qui peuvent ensuite être écrits en binaire sur un octet, c'est-à-dire une suite de 8 bits valant chacun `0` ou `1`.

Dans la table ASCII, le code associé au caractère a est 97.

1. Donner l'écriture binaire de a sur 8 bits.

Pour pouvoir corriger les erreurs durant les transmissions, on peut ajouter de la redondance dans les informations transmises, c'est-à-dire ajouter des moyens permettant de retrouver le message initial même si un ou plusieurs bits ont été modifiés. Une manière de le faire consiste à envoyer trois fois le même message.

Ainsi, même si l'une des copies se retrouve modifiée, on peut retrouver le message tant que la majorité des copies a été transmise sans erreur.

La fonction `replique` suivante implémente cette stratégie. Cette fonction prend en paramètre une liste `tab` composée de `0` et de `1`.

```
1   def replique(tab):
2     n = len(tab)
3     return [tab[i // 3] for i in range(3 * n)]
```

2. Donner le résultat de l'appel `replique([0,0,1,0,1])`.

3. Recopier et compléter les lignes 14 et 16 du code de la fonction `nb_occurrences`, donné ci-après, qui prend en paramètres une liste `tab` et un entier `i` et qui renvoie un dictionnaire qui associe, à chaque élément son nombre d'occurrences.

```
 1 def nb_occurrences(tab, i):
 2   '''
 3   Renvoie un dictionnaire qui associe, à chaque élément
 4   apparaissant dans tab entre la position 3i
 5   incluse et la position 3(i + 1) exclue,
 6   son nombre d'occurrences.
 7   >>> nb_occurrences([0, 0, 1, 1, 0, 1, 0, 1, 1], 1)
 8   {1: 2, 0: 1}
 9   '''
10   nb_occ = {}
11   for j in range(3 * i, 3 * (i + 1)):
12     x = tab[j]
13     if x in nb_occ:
14       ...
15     else:
16       ...
17   return nb_occ
```

4. Recopier et compléter à partir de la ligne 10 (le nombre de lignes et l'indentation sont suggérés mais ne sont pas obligatoires) du code de la fonction `majorite`, donné ci-après. Cette fonction prend en paramètre un dictionnaire `dict` et renvoie une clé du dictionnaire pour laquelle la valeur associée est la plus grande.

```
 1 def majorite(dict):
 2   '''
 3   Renvoie une clé du dictionnaire dict pour laquelle la
 4   valeur associée est la plus grande.
 5   Précondition : dict est un dictionnaire dont toutes
 6   les valeurs sont positives.
 7   '''
 8   cle_max = None
 9   valeur_max = -1
10   for cle in dict.keys():
11     ...
12       ...
13       ...
14   return cle_max
```

Pour transmettre 4 bits d'information, il faut envoyer 12 bits par cette méthode.

Partie B

On s'intéresse à présent à une autre solution, reposant sur l'ajout de bits de parité.

Pour transmettre 4 bits $b_3b_2b_1b_0$, on les place dans une matrice comme suit, et on complète les lignes et les colonnes par un bit de parité (`0` ou `1`) pour que chaque ligne et chaque colonne possède un nombre pair de bits valant `1`.

b_3	b_2	bit de parité ligne 1
b_1	b_0	bit de parité ligne 2
bit de parité colonne 1	bit de parité colonne 2	bit de parité total

Lorsqu'il n'y a qu'une seule erreur, elle se situe à l'intersection de la ligne et de la colonne qui possèdent un nombre impair de bits valant `1`.

5. On reçoit le tableau suivant.

`1`	`1`	`1`
`1`	`1`	`0`
`0`	`1`	`1`

   Sachant qu'une unique erreur de transmission s'est produite, recopier le tableau et entourer le bit qui a subi cette erreur (transformation d'un `0` en `1` ou d'un `1` en `0`).

On représente une telle matrice en Python par la liste de ses lignes où chaque ligne est elle-même représentée par la liste de ses bits (`0` ou `1`).

6. Écrire le code d'une fonction `erreur_colonne` qui prend en paramètre une matrice `mat` dans laquelle exactement une erreur a eu lieu lors de la transmission et qui renvoie l'indice de la colonne ayant une parité erronée.

Grâce à cette méthode, on peut transmettre 4 bits d'information en utilisant 9 bits, tout en détectant et corrigeant une unique erreur.

Partie C

Richard Hamming a mis au point une méthode qui permet d'arriver au même résultat avec seulement 7 bits transmis.

Le tableau suivant établit une correspondance entre chaque mot de 4 bits et une unique suite de 7 bits.

Code de Hamming (4, 7)				
Mot	Code associé		Mot	Code associé
0000	0000000		1000	1110000
0001	1101001		1001	0011001
0010	0101010		1010	1011010
0011	1000011		1011	0110011
0100	1001100		1100	0111100
0101	0100101		1101	1010101
0110	1100110		1110	0010110
0111	0001111		1111	1111111

On admet que ce tableau est construit de sorte que, étant donné une suite de 7 bits,

- soit elle est présente dans le tableau ;
- soit, dans le cas contraire, il existe un unique code du tableau qui ne diffère avec elle que d'un bit.

Un mot de 4 bits ayant été encodé selon cette correspondance est transmis.

7. Une unique erreur se glisse dans cette transmission, de sorte que le code reçu est `1010000`. Déterminer, en justifiant, le mot de 4 bits initial.

On souhaite écrire une fonction de correction des codes reçus.

8. Recopier et compléter les lignes 17, 20 et 25 du code de la fonction `corriger_erreur` ci-après, qui prend en paramètre la liste des entiers 0 ou 1 correspondant au code reçu et qui renvoie cette liste si c'est un code associé, ou le code associé qui ne diffère que d'un bit de celle-ci sinon.

   Exemples :

```
>>> corriger_erreur([1,1,0,1,0,0,1])
[1, 1, 0, 1, 0, 0, 1]
>>> corriger_erreur([1,0,1,0,0,0,0])
[1, 1, 1, 0, 0, 0, 0]
```

```
# liste composée de tous les codes
# associés de Hamming(4, 7).
hamming_4_7 = [
    [0,0,0,0,0,0,0], [1,1,0,1,0,0,1],
    [0,1,0,1,0,1,0], [1,0,0,0,0,1,1],
    [1,0,0,1,1,0,0], [0,1,0,0,1,0,1],
    [1,1,0,0,1,1,0], [0,0,0,1,1,1,1],
    [1,1,1,0,0,0,0], [0,0,1,1,0,0,1],
    [1,0,1,1,0,1,0], [0,1,1,0,0,1,1],
    [0,1,1,1,1,0,0], [1,0,1,0,1,0,1],
    [0,0,1,0,1,1,0], [1,1,1,1,1,1,1]]
def corriger_erreur(code_recu):
  if code_recu in hamming_4_7:
    return code_recu
  else:
    # Copie du code reçu créée par compréhension
    code = ...
    for indice in range(7):
      # Inversion du bit d'indice courant
      code[indice] = (code[indice] + 1) ...
      if code in hamming_4_7:
        return code
      else:
        # Réinit. du bit d'indice courant
        code[indice] = ...
```

On se propose de construire un décodeur pour le code de Hamming (4, 7) à l'aide d'un arbre binaire. Il s'agit d'un arbre binaire de hauteur 7 dont chaque feuille est étiquetée par le mot de 4 bits susceptible d'avoir donné le code correspondant au chemin menant à cette feuille.

Pour décoder un code reçu, on descend dans l'arbre en lisant ce code de la gauche vers la droite. Si on rencontre un bit valant `0`, on continue dans le sous-arbre gauche. Si on rencontre un bit valant `1`, on continue dans le sous-arbre droit.

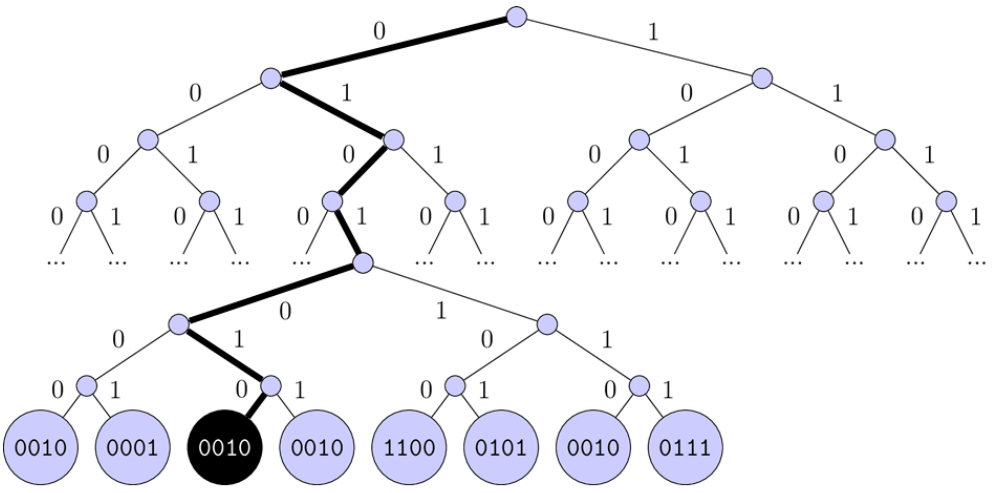

Figure 1. Représentation partielle de l'arbre décodeur du code de Hamming (4, 7).

Par exemple, le chemin indiqué en gras sur la Figure 1 indique comment on peut retrouver le mot `0010` après avoir reçu le code `0101010`.

9. Indiquer combien l'arbre décodeur complet du code de Hamming (4,7) comporte de feuilles.

Un arbre binaire non vide est représenté en Python par une classe `Noeud` qui possède 3 attributs :

- `gauche` correspond à son sous-arbre gauche s'il s'agit d'un nœud interne, et vaut `None` s'il s'agit d'une feuille ;
- `droit` correspond à son sous-arbre droit s'il s'agit d'un nœud interne, et vaut `None` s'il s'agit d'une feuille ;
- `etiquette` correspond à la chaîne de caractères désignant le mot décodé s'il s'agit d'une feuille, et vaut `None` s'il s'agit d'un nœud interne.

10. Recopier et compléter les lignes 12 à 14 du code de la fonction récursive `decode`, donné ci-après. Cette fonction prend en paramètres un arbre décodeur `arbre`, une liste `code` et un indice `i` et renvoie le mot étiquetant la feuille atteinte.

```
 1 def decode(arbre, code, i):
 2   '''
 3   Descend dans l'arbre binaire arbre en lisant le
 4   tableau code à partir de l'indice i et renvoie
 5   le mot étiquetant la feuille atteinte.
 6   Précondition : arbre est un arbre binaire
 7   de hauteur len(code) - i
 8   '''
 9   if i == len(code):
10     return arbre.etiquette
11   if code[i] == 0:
12     return decode(...)
13   if code[i] == 1:
14     return decode(...)
```

Exercice 2 (6 points)

Cet exercice porte sur la gestion des bugs, l'algorithmique, les structures de données et la programmation orientée objet.

Partie A

Un jour, Bob s'apprête à manger un collier de bonbons, et se pose la question suivante : « *Si je mange un bonbon sur trois, encore et encore jusqu'à ce qu'il n'en reste qu'un seul, quel sera le dernier bonbon restant ?* »

Figure 1. Collier de bonbons

Pour un collier ayant `5` bonbons, il décide de les numéroter de `0` à `4`. Il commence par manger le bonbon d'indice `0`, se décale de trois bonbons et mange ensuite celui d'indice 3. En répétant la démarche, il mange ensuite le bonbon d'indice 2 et enfin celui d'indice `4`.

Les indices des bonbons mangés sont donc, dans l'ordre, `0`, 3, 2 et `4`. Le bonbon restant est celui d'indice `1`.

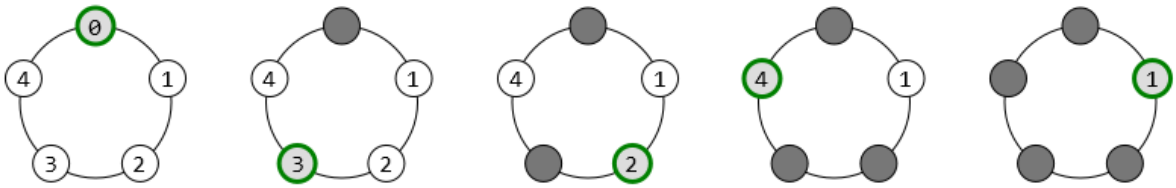

Figure 2. Les étapes pour un collier de `5` bonbons

1. Donner les indices dans l'ordre dans lequel les bonbons sont mangés dans le cas où le collier possède initialement `8` bonbons et l'indice du bonbon restant.

Afin de répondre à la question dans un cadre général, Bob décide de formaliser le problème. Il considère un collier de `n` bonbons numérotés de `0` à `n - 1`, où `n` est un entier strictement positif.

Bob vient d'étudier en classe les valeurs booléennes. Il se dit qu'il peut représenter avec Python le collier par une liste `collier` telle que, pour toute valeur entière de `i` comprise entre `0` et `n - 1`, la valeur booléenne `collier[i]` vaut `True` si le bonbon d'indice `i` du `collier` est encore présent, et vaut `False` si le bonbon d'indice `i` du `collier` a été mangé.

Dès lors, il envisage l'algorithme suivant :

- on initialise une liste `collier` représentant n bonbons qui n'ont pas encore été mangés ;
- on commence par manger le bonbon à l'indice 0 ;
- tant qu'il reste des bonbons à manger :
  - on détermine l'indice du prochain bonbon à manger dans la liste `collier` ;
  - on mange le bonbon à cet indice ;
- on renvoie l'indice du dernier bonbon mangé.

Afin de créer la liste `collier` décrite ci-dessus, Bob saisit dans la console l'instruction

```
collier = [true for i in range(8)]
```

Il obtient alors le message d'erreur suivant :

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'true' is not defined
```

2. Expliquer ce qu'est une erreur de type `NameError` et comment la corriger dans l'instruction proposée.

Bob écrit ensuite le code d'une fonction `dernier` qui prend en paramètre le nombre de bonbons n et renvoie l'indice du dernier bonbon restant. On fournit ci-après une partie du code de la fonction `dernier`.

```
 1 def dernier(n):
 2     collier = ...
 3     indice = 0
 4     collier[indice] = False
 5     for etape in range(n-1):
 6         nb_bonbons_vus = 0
 7         while nb_bonbons_vus ...:
 8             indice += 1
 9             if ...:
10                 indice = 0
11             if ...:
12                 nb_bonbons_vus += 1
13         collier[indice] = ...
14     return indice
```

3. Recopier et compléter les lignes 2, 7, 9, 11 et 13 du code de la fonction `dernier`.

Partie B

Bob se dit qu'une structure de file lui permettrait de résoudre astucieusement le problème des bonbons.

On considère la classe `File` dont on fournit ci-après l'interface.

```
 1 class File:
 2     """Classe définissant une structure de file"""
 3
 4     def __init__(self):
 5         """Initialise une file vide"""
 6
 7     def est_vide(self):
 8         """Renvoie le booléen indiquant
 9          si la file est vide"""
10
11     def enfile(self, x):
12         """Place x à la queue de la file"""
13
14     def defile(self):
15         """Retire et renvoie l'élément placé à la
16         tête de la file
17         Provoque une erreur si la file est vide
18         """
19
20     def affiche(self):
21         """Affiche la file"""
```

Le code Python ci-après montre un exemple d'utilisation de la classe `File`.

```
>>> f = File()
>>> f.enfile(0)
>>> f.enfile(1)
>>> f.affiche()
(Tête) 0 1 (Queue)
```

L'acronyme LIFO signifie « Last In First Out » à savoir « Dernier entré, premier sorti ». L'acronyme FIFO signifie « First In First Out » à savoir « Premier entré, premier sorti ».

4. Donner l'acronyme le plus adapté à la structure de donnée `File`.

5. Déterminer l'affichage réalisé lors de l'exécution des instructions ci-après.

```
>>> f = File()
>>> for x in [0, 1, 2, 3, 4]:
>>>     f.enfile(x)
>>> f.defile()
>>> f.enfile(f.defile())
>>> f.enfile(f.defile())
>>> f.affiche()
```

6. Écrire le code de la fonction `dernier_file` qui prend en paramètre le nombre de bonbons n et renvoie l'indice du dernier bonbon restant.

Partie C

Bob souhaite utiliser la structure de données « *liste doublement chaînée* ». Une telle liste est composée de *maillons* contenant chacun trois informations :

- une valeur ;
- un prédécesseur et un successeur qui sont tous deux des maillons.

Cette structure se prête bien au problème des bonbons : dans un collier, un bonbon est précédé et suivi par d'autres bonbons. Le successeur du dernier bonbon est le premier et le prédécesseur du premier, le dernier.

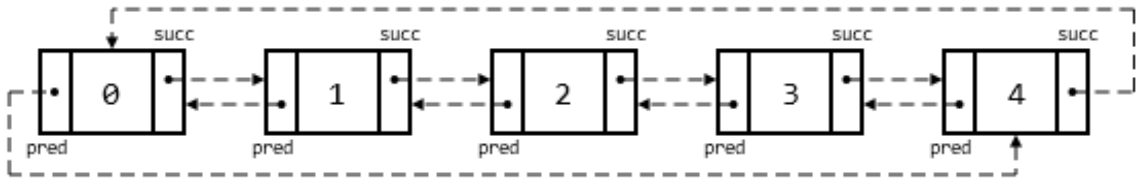

Figure 3. Collier de cinq `Bonbon`

Bob crée la classe `Bonbon` ci-après :

```
1 class Bonbon:
2     def __init__(self, valeur):
3         self.pred = None  # prédécesseur de ce bonbon
4         self.valeur = valeur
5         self.succ = None  # successeur de ce bonbon
```

7. Donner le terme correspondant aux variables `pred`, `valeur` et `succ` dans le vocabulaire de la programmation orientée objet.

Les instructions ci-dessous permettent de représenter un collier de trois bonbons de valeurs `0`, `1` et `2`.

```
>>> zero = Bonbon(0)
>>> un = Bonbon(1)
>>> deux = Bonbon(2)

>>> zero.succ = un
>>> un.pred = zero

>>> un.succ = deux
>>> deux.pred = un

>>> deux.succ = zero
>>> zero.pred = deux

>>> a = zero.succ.valeur
>>> b = un.succ.succ.pred.valeur
```

8. Déterminer les valeurs des variables a et b après l'exécution de ces instructions.

La fonction `creer_collier` prend en paramètre un entier `n` strictement positif représentant la taille d'un collier et renvoie un objet de type `Bonbon` représentant le premier bonbon (de valeur `0`) du collier.

On prendra soin de faire se succéder et précéder les différents bonbons ainsi que de « refermer » le collier en liant le dernier bonbon au premier.

9. Recopier et compléter les lignes 5, 6, 7, 9 et 10 du code de la fonction `creer_collier`, donné ci-après.

```
 1 def creer_collier(n):
 2     premier = Bonbon(0)
 3     actuel = premier
 4     for i in range(1, n):
 5         nouveau = Bonbon(...)
 6         actuel.succ = ...
 7         ...
 8         actuel = nouveau
 9     ...
10     ...
11     return premier
```

On considère le code Python suivant.

```
>>> bonbon = Bonbon(3)
>>> bonbon.pred = bonbon
>>> bonbon.succ = bonbon
```

À l'issue de l'exécution de ce code, on obtient la liste doublement chaînée représentée ci-dessous.

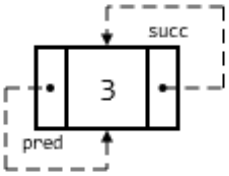

Figure 4. Un collier d'un seul `Bonbon`

10. On considère le code Python suivant.

    ```
    >>> premier = creer_collier(4)
    >>> premier.pred.succ = premier.succ
    >>> premier.succ.pred = premier.pred
    >>> bonbon = premier.succ
    ```

    Dessiner une représentation du « collier » dont le premier élément est l'objet `bonbon` obtenu à l'issue de l'exécution du code Python ci-dessus.

11. Dans le cas où il ne reste qu'un bonbon, donner l'expression qui s'évalue à `True`, parmi les quatre propositions ci-dessous :

    - Proposition A : `valeur.succ == valeur.bonbon`
    - Proposition B : `pred == succ`
    - Proposition C : `bonbon.valeur == bonbon.succ.valeur`
    - Proposition D : `bonbon.valeur == succ.valeur`

12. Recopier et compléter les lignes 3, 4, 5 et 6 du code de la fonction `dernier_chaine`, donné ci-après, qui prend en paramètre le nombre de bonbons n et renvoie la valeur du dernier bonbon restant.

    ```
    1 def dernier_chaine(n):
    2     bonbon = creer_collier(n)
    3     while ... != ...:
    4         bonbon.pred.succ = ...
    5         ... = bonbon.pred
    6         bonbon = ... # décalage de 3 bonbons
    7     return bonbon.valeur
    ```

Exercice 3 (8 points)

Cet exercice porte sur les bases de données, la programmation en Python, la récursivité et les algorithmes de parcours de graphes.

Partie A

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`.

La compagnie aérienne *AirInfo* souhaite utiliser un système de gestion de bases de données relationnelles afin de l'aider à gérer les informations dont elle dispose sur les aéroports desservis, les vols proposés, les passagers et les réservations effectuées par ces passagers.

Elle crée donc la base de données `compagnie_aerienne`, constituée de quatre relations. Le schéma relationnel de cette base de données est le suivant :

- `aeroport`(`id_aeroport` : `TEXT`, `ville` : `TEXT`, `pays` : `TEXT`)
- `vol`(`id_vol` : `TEXT`, `aeroport_dep` : `TEXT`, `aeroport_arr` : `TEXT`, `distance` : `INT`)
- `passager`(`id_passager` : `INT`, `nom` : `TEXT`, `prenom` : `TEXT`, `ville` : `TEXT`, `d_totale` : `INT`)
- `reservation`(`id_vol` : `TEXT`, `id_passager` : `INT`)

Une clé primaire de la relation `vol` est l'attribut `id_vol`. Les autres clés primaires et étrangères ne sont pas précisées.

Les quatre tables ci-après constituent, en l'état, la base de données `compagnie_aerienne` complète.

Les valeurs des champs `distance` et `d_totale` intervenant respectivement dans les tables `vol` et `passager` sont exprimées en milliers de kilomètres.

aeroport		
id_aeroport	ville	pays
CDG	Paris	France
IAD	Washington	USA
QPP	Berlin	Allemagne
NRT	Tokyo	Japon
SYD	Sydney	Australie
YUL	Montréal	Canada

vol			
id_vol	aeroport_dep	aeroport_arr	distance
AI0006	CDG	IAD	6
AI0015	IAD	CDG	6
AI0256	CDG	SYD	17
AI0258	SYD	CDG	17
AI0276	CDG	NRT	10
AI0292	NRT	CDG	10
AI1280	NRT	QPP	9
AI1681	QPP	NRT	9
AI1785	NRT	SYD	8
AI1845	SYD	NRT	8

passager				
id_passager	nom	prenom	ville	d_totale
1	Dupont	Alice	Paris	6
2	Smith	John	Washington	6
3	Brown	Sarah	Berlin	9
4	Yamada	Taro	Tokyo	17
5	Williams	Emma	Sydney	10

reservation	
id_vol	id_passager
AI0006	1
AI0006	2
AI0256	4
AI0276	5
AI1681	3

1. Expliquer pourquoi l'attribut `id_vol` ne peut pas être une clé primaire de la table `reservation`.

2. Proposer une clé primaire pour la table `reservation`.

3. Expliquer le rôle d'une clé étrangère dans une relation.

4. Écrire le résultat renvoyé par la requête SQL suivante :

```
SELECT id_vol
FROM vol
WHERE aeroport_arr = 'CDG';
```

5. Écrire une requête SQL qui donne les noms des villes qui sont destination d'un vol au départ de l'aéroport CDG.

La compagnie *AirInfo* envisage de récompenser la fidélité de ses usagers en tenant compte de la distance totale qu'ils parcourent sur leurs lignes. Ainsi, à chaque nouvelle réservation, la table `passager` doit être mise à jour : si l'usager avait déjà parcouru une distance totale d_{totale}, puis réserve un vol d'une distance d_{vol}, l'attribut `d_totale` est modifié en prenant pour nouvelle valeur $d_{\text{totale}} + d_{\text{vol}}$.

6. La passagère d'identifiant 5 a déjà parcouru 10 milliers de kilomètres. Elle réserve un vol de Washington (IAD) à Paris (CDG), d'une distance de 6 milliers de kilomètres.

   Écrire la requête permettant de mettre à jour la table `passager`.

La compagnie aérienne offre maintenant la possibilité de relier Paris CDG à Montréal YUL (Canada) par un vol de 6 milliers de kilomètres. Afin de prendre en compte ce nouveau trajet dans la base de données, on souhaite l'ajouter dans la table `vol` dont la clé primaire est `id_vol`. On propose, de manière incorrecte, la requête d'insertion ci-après :

```
INSERT INTO vol
VALUES ('AI0256', 'CDG', 'YUL', 6);
```

7. Proposer, en justifiant, une correction relative à l'erreur commise.

Partie B

Dans cette partie, on considère les villes suivantes : Washington (W) ; Paris (P) ; Berlin (B) ; Tokyo (T) ; Sydney (S).

Les distances des vols entre ces villes, exprimées en milliers de kilomètres, sont les suivantes :

- Washington - Paris : 6 ;
- Washington - Berlin : 7 ;
- Paris - Berlin : 1 ;
- Paris - Tokyo : 10 ;
- Paris - Sydney : 17 ;
- Berlin - Tokyo : 9 ;
- Tokyo - Sydney : 8.

La compagnie *AirInfo* propose certains de ces vols. Son réseau aérien est représenté par le graphe pondéré non orienté de la Figure 1 dans lequel :

- chaque ville est représentée par un sommet ;
- chaque vol direct entre deux villes est représenté par une arête.

Le nombre indiqué sur chaque arête est appelé poids : il représente la distance entre deux villes. Cette distance est la même dans un sens ou dans l'autre.

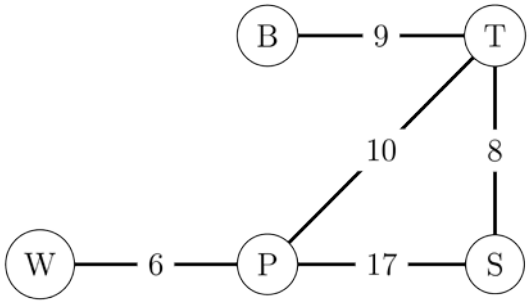

Figure 1. Graphe non orienté pondéré correspondant au réseau aérien de la compagnie *AirInfo*

On choisit de représenter ce graphe en Python à l'aide d'un dictionnaire dans lequel chaque clé est un sommet du graphe et la valeur associée à cette clé est elle-même un dictionnaire, dont les clés sont les villes voisines et les valeurs sont les distances correspondantes.

Le dictionnaire associé au graphe, représenté par la Figure 1, est donné ci-après.

```
graphe_airinfo = {
    'W': {'P': 6},
    'P': {'W': 6, 'T': 10, 'S': 17},
    'B': {'T': 9},
    'T': {'B': 9, 'P': 10, 'S': 8},
    'S': {'P': 17, 'T': 8}
}
```

8. Déterminer la valeur de l'expression `graphe_airinfo['T']['P']`.

9. Écrire le code d'une fonction Python `vol_direct` permettant de déterminer s'il existe une liaison directe entre deux villes. Cette fonction prend en paramètres un dictionnaire `graphe` représentant le graphe, et deux clés du dictionnaire `ville1` et `ville2` représentant deux villes, et renvoie `True` si une telle liaison entre les villes `ville1` et `ville2` existe, c'est-à-dire si les deux sommets `ville1` et `ville2` sont reliés dans le graphe `graphe` par une arête, `False` sinon.

   Exemples :

   ```
   >>> vol_direct(graphe_airinfo, 'T', 'B')
   True
   >>> vol_direct(graphe_airinfo, 'W', 'B')
   False
   ```

Afin de limiter leurs empreintes carbone, les voyageurs demandent fréquemment aux compagnies aériennes de déterminer, à partir d'une ville donnée, la liste des villes qu'il est possible d'atteindre par un vol direct tout en ne dépassant pas une certaine distance.

10. Écrire le code d'une fonction Python `liste_villes_proches` qui permet de répondre à la demande des voyageurs. Cette fonction prend en paramètres un dictionnaire `graphe` représentant le graphe, une clé du dictionnaire `ville` et un entier `d_max` représentant une distance et renvoie la liste des villes reliées à `ville` par une arête dans `graphe` dont le poids est au plus égal à `d_max`.

    Exemples :

    ```
    >>> liste_villes_proches(graphe_airinfo, 'T', 7)
    []
    >>> liste_villes_proches(graphe_airinfo, 'T', 9)
    ['B', 'S']
    ```

La compagnie aérienne *Droidevant*, concurrente de la compagnie *AirInfo*, assure des liaisons différentes. Son réseau aérien peut également être représenté par un graphe dont le dictionnaire correspondant en Python est donné ci-après.

```
graphe_droidevant = {
    'W': {'P': 6, 'B': 7},
```

```
3     'P': {'W': 6, 'B': 1},
4     'B': {'W': 7, 'P': 1},
5     'T': {'S': 8},
6     'S': {'T': 8}
7 }
```

11. Dessiner, en indiquant le poids sur chaque arête, le graphe représentant le réseau aérien de la compagnie *Droidevant*.

On dit qu'un graphe est connexe s'il existe un chemin entre chaque paire de sommets, autrement dit, si pour tout sommet `s1` et pour tout sommet `s2`, il existe un chemin entre `s1` et `s2` dans le graphe.

12. Indiquer, parmi les deux propositions suivantes, celle qui est correcte :

    - Proposition A : le graphe de la compagnie *AirInfo* est connexe ;
    - Proposition B : le graphe de la compagnie *Droidevant* est connexe.

On peut adapter un algorithme de parcours de graphe pour déterminer si un graphe est connexe.

On considère la fonction `parcours` qui permet d'explorer les villes d'un graphe accessibles à partir d'une ville donnée et qui prend en paramètres :

- `graphe` : un dictionnaire représentant le graphe ;
- `visitees` : une liste des villes déjà visitées ;
- `ville` : la ville actuelle à partir de laquelle on effectue l'exploration.

```
1 def parcours(graphe, visitees, ville):
2     """Parcours d'un graphe à partir d'une ville non
3     visitée, en ayant déjà visité un certain nombre de
4     villes.
5     """
6     # Marque la ville comme visitée
7     visitees.append(ville)
8     # Parcourt les voisines de la ville
9     for voisine in graphe[ville]:
10        if voisine not in visitees:
11            # Explore depuis les voisines non visitées
12            parcours(graphe, visitees, voisine)
```

13. Expliquer en quoi la fonction `parcours` est une fonction récursive.

14. Déterminer le contenu des variables `visitees1` et `visitees2` après l'exécution des lignes de code données ci-après.

```
1 visitees1 = []
2 parcours(graphe_airinfo, visitees1, 'W')
3
4 visitees2 = []
5 parcours(graphe_droidevant, visitees2, 'W')
```

15. Indiquer, parmi les propositions suivantes, laquelle correspond au type de parcours effectué par la fonction `parcours` :

    - Proposition A : un parcours en largeur ;
    - Proposition B : un parcours en grandeur ;
    - Proposition C : un parcours en profondeur.

On cherche à écrire une fonction `est_connexe` qui prend en paramètre un graphe, représenté à l'aide d'un dictionnaire de dictionnaires de voisins, et qui renvoie `True` si le graphe est connexe, `False` sinon.

On admet disposer d'une fonction `ville_arbitraire` qui renvoie un sommet arbitraire d'un graphe.

Par exemple, `ville_arbitraire(graphe_airinfo)` pourrait renvoyer `'T'` ou n'importe quel autre sommet.

On considère le code de la fonction Python incomplet suivant :

```
1 def est_connexe(graphe):
2     """Vérifie si un graphe est connexe."""
3     depart = ville_arbitraire(graphe)
4     visitees = ...
5     parcours(graphe, visitees, depart)
6     return ...
```

16. Recopier et compléter les lignes 4 et 6 du code de la fonction `est_connexe`. On pourra, si nécessaire, ajouter de nouvelles lignes.

On considère maintenant le code de la fonction, donné ci-après, `mystere`, qui prend en paramètres :

- `graphe`, un dictionnaire représentant un graphe ;
- `ville`, une clé du dictionnaire `graphe` représentant une ville de départ ;
- `chemin`, une liste de clés représentant les villes ;
- `cout`, un entier représentant une distance ;
- `arrivee`, une clé du dictionnaire `graphe` représentant une ville d'arrivée.

```
 1 def mystere(graphe, ville, chemin, cout, arrivee):
 2     # Ajoute la ville actuelle au chemin
 3     chemin = chemin + [ville]
 4     if ville == arrivee:
 5         # Affiche le chemin et son coût total
 6         print(chemin, cout)
 7     # Parcourt les villes voisines et leurs poids
 8     for voisine, poids in graphe[ville].items():
 9         # Vérifie que la ville n'est pas déjà visitée
10         if voisine not in chemin:
11             mystere(graphe, voisine, chemin, cout + poids,
arrivee)
```

17. Déterminer les affichages produits lors de l'exécution de l'appel suivant :

    ```
    mystere(graphe_airinfo, 'W', [], 0, 'B')
    ```

18. Expliquer, de manière générale, ce que réalise l'appel `mystere(graphe, ville, [], 0, arrivee)` pour un graphe `graphe` et deux villes `ville` et `arrivee`.