BACCALAURÉAT GÉNÉRAL

ÉPREUVE D'ENSEIGNEMENT DE SPÉCIALITÉ

Mars 2026 – Bac Blanc

N.S.I.
Numérique et Sciences Informatiques

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 à 17

Exercice 1 (6 points)

Cet exercice porte sur les arbres binaires, la récursivité et la programmation orientée objet.

Cet exercice porte sur l'identification de végétaux (tilleul, ficus, …) à partir de caractéristiques de leurs *folia* (nom scientifique des feuilles d'un végétal) : simples ou complexes, disposées de façon alternée ou non, etc.

Par exemple, un tilleul a des *folia* simples, disposées de façon alternée mais pas en hélice, en forme de cœur et à bord denté. Un ficus a également des *folia* simples et disposées de façon alternée. Cependant elles sont insérées en hélice et sont de forme ovale. Un robinier a des *folia* complexes, disposées de façon alternée et non dentées.

Pour identifier un végétal à l'aide des caractéristiques de ses *folia*, on utilise un arbre binaire appelé **arbre de décision**. Un exemple de tel arbre de décision est partiellement représenté sur la figure 1 ci-dessous (les parties non représentées de cet arbre sont indiquées par des points de suspension).

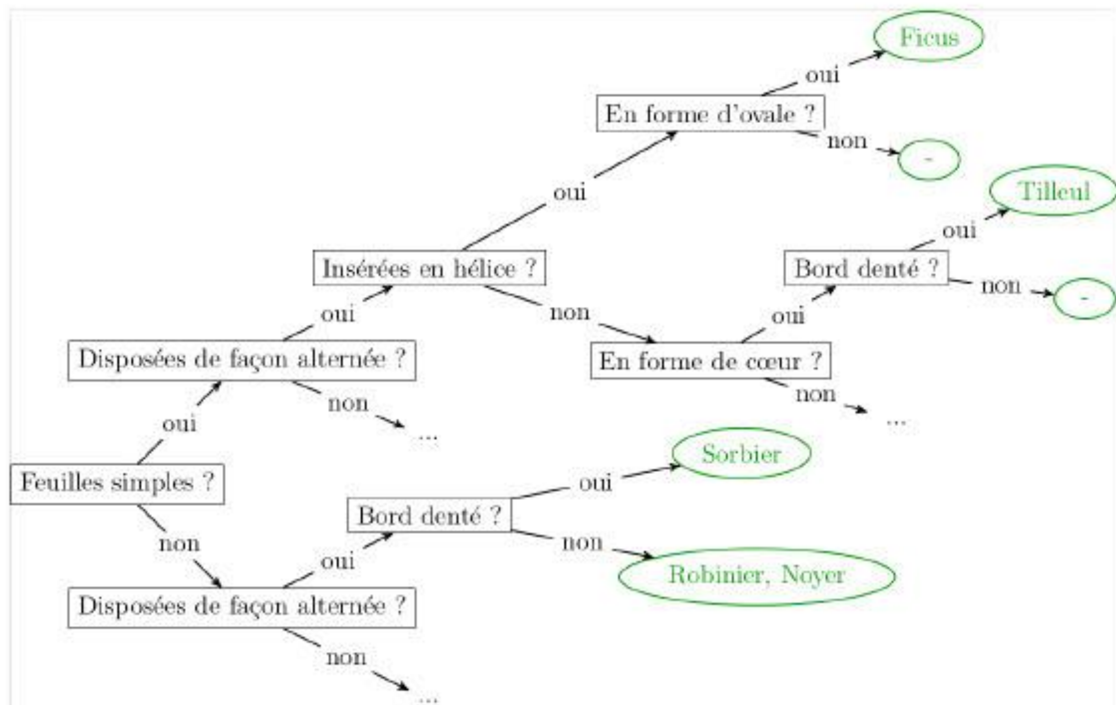

Figure 1. Extrait d'un arbre de décision aidant à reconnaître un végétal à partir des caractéristiques de ses *folia*.

Les rectangles sont les **nœuds** de l'arbre de décision. Ils correspondent chacun à une question. Les ovales sont les **feuilles** de l'arbre de décision. Ils correspondent chacun à un ensemble de végétaux. Pour chaque question, il est possible de répondre par oui ou par non ce qui permet d'atteindre soit un nouveau nœud, c'est-à-dire une nouvelle question, soit une feuille de l'arbre de décision. Cette feuille contient le plus souvent

un seul végétal, éventuellement plusieurs si leurs *folia* ont les mêmes caractéristiques, et éventuellement aucun si aucun végétal connu ne présente ces caractéristiques. Par exemple, robinier et noyer ont tous les deux des *folia* complexes (non simples), disposées de façon alternée et non dentées : ils sont donc dans la même feuille de l'arbre de décision de la figure 1.

1. On observe un végétal dont les *folia* sont complexes (non simples), disposées de façon alternée et à bord denté. D'après l'arbre de décision de la figure 1, peut-on identifier ce végétal ? Si oui, quel est-il ?

2. On observe un végétal dont les *folia* sont simples, disposées de façon alternée, insérées en hélice et ne sont pas de forme d'ovale. D'après l'arbre de décision de la figure 1, peut-on identifier ce végétal ? Si oui, quel est-il ?

L'arbre de décision est représenté en langage Python en utilisant une classe `Noeud` et une classe `Feuille_resultat` dont les définitions sont données ci-dessous.

```
1 class Noeud:
2     def __init__(self, question, sioui, sinon):
3         self.question = question
4         self.sioui = sioui
5         self.sinon = sinon
```

```
1 class Feuille_resultat:
2     def __init__(self, vegetaux):
3         self.vegetaux = vegetaux
```

La classe `Noeud` a trois attributs :

- un attribut `question`, qui est une chaîne de caractères représentant une question ;
- un attribut `sioui`, qui peut être soit un objet de la classe `Noeud` représentant une autre question, soit un objet de la classe `Feuille_resultat` ;
- un attribut `sinon`, qui peut être soit un objet de la classe `Noeud` représentant une autre question, soit un objet de la classe `Feuille_resultat`.

La classe `Feuille_resultat` a un seul attribut, `vegetaux`, qui est une liste (éventuellement vide) de chaînes de caractères, dans laquelle chaque chaîne est le nom d'un végétal.

Par exemple, pour l'arbre de décision de la figure 1, pour le `Noeud` dont la question est `'En forme d'ovale?'`, l'attribut `sioui` est un objet de la classe `Feuille_resultat` dont l'attribut `vegetaux` est la liste `['Ficus']` alors que l'attribut `sinon` de ce nœud est un objet de la classe `Feuille_resultat` dont l'attribut `vegetaux` est la liste vide.

3. Écrire en langage Python le code permettant de construire l'arbre de décision de la **figure 2** ci-dessous et de l'affecter à une variable nommée `arbre_2`.

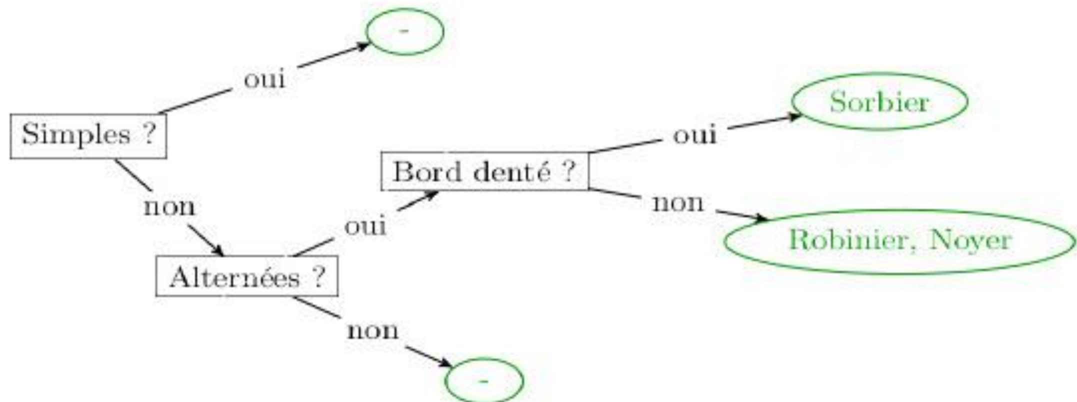

Figure 2. Arbre de décision 2.

On souhaite écrire une méthode `est_resultat` pour chacune des classes `Noeud` et `Feuille_resultat`. Un objet de la classe `Feuille_resultat` est un résultat, la méthode doit renvoyer `True`. Un objet de la classe `Noeud` n'est pas un résultat, la méthode doit renvoyer `False`.

4. Écrire le code de la méthode `est_resultat` pour la classe `Noeud`.

5. Écrire le code de la méthode `est_resultat` pour la classe `Feuille_resultat`.

On souhaite connaître le nombre de végétaux identifiables par un arbre de décision.

6. Écrire le code de la méthode `nb_vegetaux` pour la classe `Feuille_resultat`.

7. Écrire le code de la méthode `nb_vegetaux` pour la classe `Noeud`, qui prend en compte tous les végétaux identifiables à partir de ce nœud.

On souhaite enfin écrire une méthode `liste_questions` pour chacune des classes `Noeud` et `Feuille_resultat` afin d'obtenir la liste des questions présentes dans un arbre de décision. L'ordre des éléments dans cette liste n'a pas d'importance, de plus elle peut contenir des doublons. On remarque que :

- si `f` est un objet de la classe `Feuille_resultat`, alors `f.liste_questions()` est la liste vide ;

- le résultat de l'appel `arbre_2.liste_questions()` est la liste `['Simples ?', 'Alternées? ', 'Bord denté ?']` (ou une liste avec les mêmes éléments mais dans un ordre différent).

8. Écrire le code de la méthode `liste_questions` pour la classe `Feuille_resultat`.

9. Écrire le code de la méthode `liste_questions` pour la classe `Noeud`, qui prend en compte toutes les questions accessibles à partir de ce nœud. On rappelle que l'opérateur + en Python permet de concaténer des listes, par exemple la valeur de l'expression `[1,2]+[3,4,5]` est la liste `[1,2,3,4,5]`.

Pour représenter les caractéristiques des *folia* d'un végétal, on utilise un dictionnaire. Les clés du dictionnaire sont les questions de l'arbre de décision et les valeurs sont `True` ou `False` selon la réponse.

Par exemple, le dictionnaire décrivant les *folia* du sorbier pour l'arbre de décision de la figure 2 est le dictionnaire `folia_sorbier` défini ci-dessous.

```
1 folia_sorbier = {
2   'Simples ?': False,
3   'Alternées ?': True,
4   'Bord denté ?': True
5 }
```

En revanche le dictionnaire `folia_tilleul` ci-dessous qui décrit (partiellement) les *folia* du tilleul n'est pas adapté pour l'arbre de décision de la figure 1 car des données sont manquantes. Par exemple, la question `'Feuilles simples ?'` n'est pas une clé de ce dictionnaire alors que c'est une question présente dans l'arbre.

```
1 folia_tilleul = {
2   'En forme d'ovale ?': False,
3   'Disposées de façon alternée ?': True,
4   'Bord denté ?': True
5 }
```

On cherche à éviter ce genre de cas, afin de ne pas d'utiliser un arbre de décision pour classifier un végétal à partir d'un dictionnaire qui n'est pas assez renseigné.

10. Écrire une fonction `est_bien_renseigne` qui prend en paramètres :

    – un dictionnaire `dico_vegetal` qui donne les caractéristiques des *folia* d'un végétal ,
    – un arbre de décision représenté par un objet `arbre` de la classe `Feuille_resultat` ou de la classe `Noeud`,

et qui renvoie `True` si toutes les questions présentes dans `arbre` sont des clés de `dico_vegetal`.

11. Écrire une fonction `identifier_vegetaux` qui prend en paramètres :

    – un dictionnaire `dico_vegetal` qui donne les caractéristiques des *folia* d'un végétal,
    – un arbre de décision représenté par un objet `arbre` de la classe `Feuille_resultat` ou de la classe `Noeud`,

    et qui renvoie la liste, éventuellement vide, des noms des végétaux dont les *folia* correspondent aux caractéristiques du dictionnaire.

    Par exemple l'appel `identifier_vegetaux(arbre_2,folia_sorbier)` devra renvoyer la liste `['Sorbier']`.

    On suppose que toutes les questions de l'arbre de décision `arbre` apparaissent comme des clés dans le dictionnaire `dico_vegetal`.

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`.

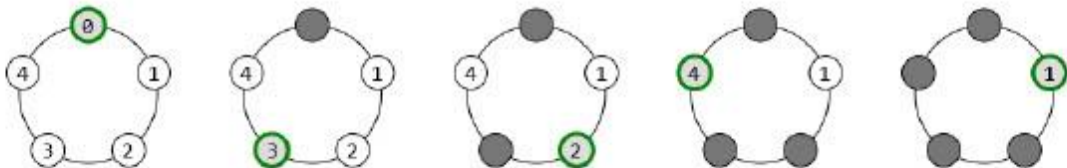

Figure 2. Les étapes pour un collier de `5` bonbons

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.

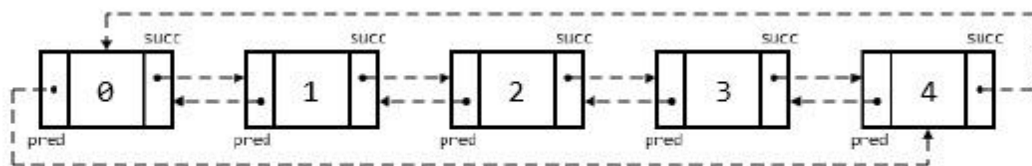

Figure 3. Collier de cinq `Bonbon`

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.

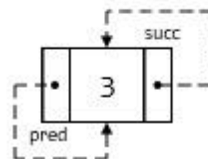

Figure 4. Un collier d'un seul `Bonbon`

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 la programmation objet en langage Python, les graphes et les bases de données.

Partie A

Nous avons représenté un parc d'attractions par un graphe. Les sommets de ce graphe sont des attractions. Chaque attraction a une durée (en minutes). Les arêtes de ce graphe représentent la durée (en minutes) pour aller d'une attraction à une autre. Dans ce parc d'attractions, toutes les attractions ont des noms uniques.

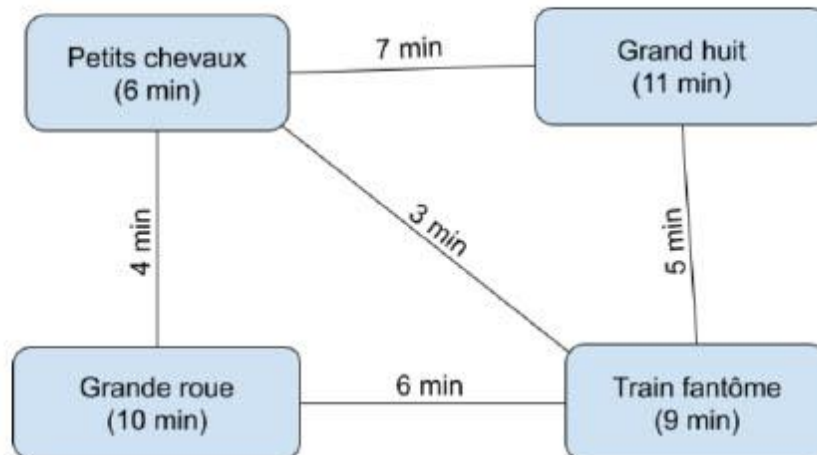

Figure 1. Parc d'attractions

Les attractions sont représentées par des objets de la classe `Attraction` dont le code est donné ci-dessous.

```
1 class Attraction:
2     def __init__(self, nom, duree):
3         self.nom = nom
4         self.duree = duree
5         self.voisines = []
```

Le graphe précédent peut être représenté, d'une façon incomplète, en langage Python ainsi :

```
1 a1 = Attraction("Grand huit", 11)
2 a2 = Attraction("Petits chevaux", 6)
3 a3 = Attraction("Train fantôme", 9)
4 a4 = Attraction("Grande roue", 10)
5 a1.voisines = [(a2,7), (a3,5)]
6 a2.voisines = [(a1,7), (a3,3), (a4,4)]
7 a3.voisines = [(a1,5), (a2,3), (a4,6)]
8 a4.voisines = ...
```

Par mesure de sécurité, les gérants du parc d'attractions ont ralenti la vitesse de rotation de la grande roue. Sa durée est maintenant de 12 minutes.

1. En considérant la modélisation du parc d'attractions ci-dessus, écrire une ligne de code permettant de faire cette modification.

2. Donner et expliquer la valeur de l'expression `a2.voisines[2][1]`.

3. Expliquer la ligne 7 de ce code.

4. Recopier et compléter la ligne 8 de ce code.

5. Expliquer pourquoi cette modélisation du parc d'attractions est réalisée avec un graphe non orienté.

Pour faciliter la gestion du parc d'attractions, ses dirigeants proposent aux usagers des *balades* dans le parc. Une *balade* est un chemin du graphe représentant le parc d'attractions. Les usagers choisissant une balade doivent faire les attractions dans l'ordre de parcours du chemin. La durée d'une balade est la durée totale pour parcourir la balade, c'est-à-dire la somme des durées de ses sommets et de ses arêtes.

En langage Python, on modélise une balade par un tableau de sommets du graphe. Par exemple, le tableau `[a1, a2, a3, a1, a3]` est une balade du graphe précédent.

6. Calculer la durée en minutes de la balade représentée par le tableau `[a1, a2, a3]` et expliquer le calcul effectué en une phrase.

7. Expliquer pourquoi le tableau `[a2, a1, a4, a3]` n'est pas une balade du parc d'attractions.

On considère qu'il est possible de comparer des objets de la classe `Attraction` entre eux à l'aide de l'opérateur ==.

8. Écrire une fonction `sont_voisines` qui prend comme arguments deux attractions de la classe `Attraction` et qui renvoie `True` si ces deux attractions sont voisines et `False` sinon.

9. Écrire une fonction `est_balade` qui prend comme argument un tableau de sommets de type `Attraction` et qui renvoie `True` si ce tableau est une balade et `False` sinon.

Les gérants du parc d'attractions souhaitent automatiser la création de balades, de telle sorte que désormais chaque attraction apparaisse au maximum une fois dans la balade. Pour cela, ils proposent de faire un parcours de graphe à partir d'une des attractions du parc, avec un tableau pouvant représenter une balade en paramètre. Pendant le parcours du graphe, si une attraction est atteignable depuis la dernière attraction placée dans la balade, alors elle est ajoutée à la balade.

Le code suivant est proposé :

```
1 def parcours(attr, deja_vues, balade, nb):
2     if not attr.nom in deja_vues:
3         deja_vues[attr.nom] = True
4         if nb == 0 or sont_voisines(attr, balade[nb-1]):
5             balade[nb] = attr
6             nb = nb + 1
7         for voisine in attr.voisines:
8             nb = parcours(voisine[0], deja_vues, balade, nb)
9     return nb
```

10. Donner le type de parcours effectué par la fonction `parcours` ci-dessus.

Chaque attraction apparaît au maximum une fois dans une balade. Ainsi, un tableau représentant la balade peut être initialisé à `[None, None, None, None]` si le parc d'attractions n'a que 4 attractions. Si à l'issue du parcours, les attractions n'ont pas été toutes utilisées, il sera possible de créer une copie partielle du tableau contenant uniquement les éléments différents de `None`.

11. Déterminer ce que contient le tableau `balade` après l'exécution du code ci-dessous, en utilisant les variables `a1`, `a2`, `a3` et `a4` :

```
>>> balade = [ None for _ in range(4) ]
>>> parcours(a4, {}, balade, 0)
```

12. Déterminer maintenant ce que contient le tableau `tableau` après l'exécution du code ci-dessous :

```
>>> a2.voisines = [(a1,7), (a3,3)]
>>> a4.voisines = [(a3,6)]
>>> tableau = [ None for _ in range(4) ]
>>> parcours(a3, {}, tableau, 0)
```

13. Déduire des appels à la fonction `parcours` le nom de la structure de données utilisée pour la variable `deja_vues` et expliquer en une phrase son rôle.

Partie B

Dans cette partie de l'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`), `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` .

Les visiteurs qui sont d'accord reçoivent un bracelet magnétique à l'entrée du parc d'attractions. Ce bracelet permet de les identifier et de les prendre en photos à différents points clés des attractions. Ces photos leur sont ensuite proposées à la vente. Le système est calibré pour ne pas prendre de photos des personnes ne le souhaitant pas. Les données personnelles associées sont stockées en France et les utilisateurs disposent, conformément à la loi, d'un droit de consultation, de retrait et de rectification.

Pour gérer ces photos et leur vente, le parc d'attractions utilise une base de données. La Figure 2 présente une représentation des trois relations de cette base dont les clés primaires sont les attributs soulignés, appelés `id` dans chaque relation et dont les clés étrangères sont précédées d'un caractère `#`. Pour chaque attribut est indiqué le nom de l'attribut, et son type après le symbole `:` le type `int` représente des entiers, le type `text` des chaînes de caractères et le type `float` des nombres flottants.

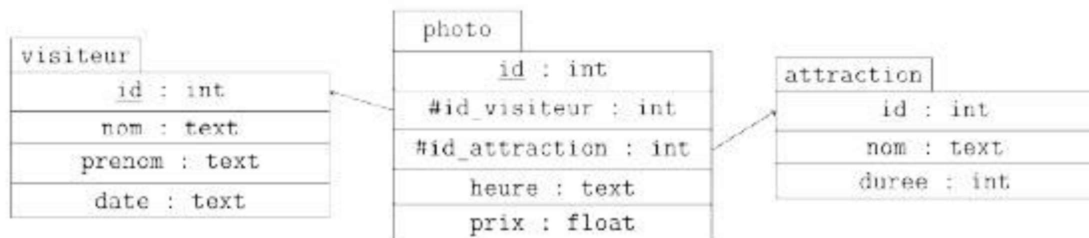

Figure 2. Représentation des relations de la base de données utilisée.

L'attribut `date` de la relation `visiteur` est une chaîne de caractères au format `'année-mois-jour'`, l'année étant écrite sur 4 chiffres, le mois sur 2 chiffres et le jour sur 2 chiffres. Par exemple, le 1er février 2025 sera représenté par la chaîne de caractères `'2025-02-01'`. L'attribut `heure` de la relation `photo` est une chaîne de caractères au format `'heures:minutes'`, en utilisant 2 chiffres pour les heures et 2 chiffres pour les minutes. Par exemple, l'heure 5 heures 49 minutes sera représentée par `'05:49'`.

14. Expliquer ce qu'est une clé primaire, puis ce qu'est une clé étrangère.

15. Écrire une requête en langage SQL qui permet d'obtenir les noms et prénoms des visiteurs présents le 11 janvier 2025 sans doublons.

En langage SQL, les opérateurs de comparaison classiques peuvent être utilisés pour comparer des chaînes de caractères entre elles. Par exemple, la condition '2025-01-01' > '2024-01-01' serait évaluée à vrai.

La fonction d'agrégation `SUM` permet de renvoyer la somme des valeurs d'un attribut. Par exemple, le code ci-dessous permet de déterminer le prix total des photos de la relation `photo` :

```
SELECT SUM(prix)
FROM photo;
```

Un visiteur, Alan TURING, est venu plusieurs fois dans le parc d'attractions en 2024. À chaque visite, il a acheté toutes les photos proposées.

16. Écrire une requête en langage SQL qui permet d'obtenir la somme totale de ce qu'Alan TURING a payé pour des photos au parc d'attractions en 2024.

Suite à un problème technique, les gérants ont utilisé la requête suivante :

```
SELECT visiteur.nom, prenom
FROM visiteur JOIN photo
  ON visiteur.id = photo.id_visiteur
JOIN attraction
  ON attraction.id = photo.id_attraction
WHERE attraction.nom = 'Grande roue'
  AND heure = '12:34'
  AND date = '2024-07-26';
```

17. Expliquer ce qu'ils voulaient savoir.

Les gérants du parc d'attractions décident d'étoffer leur offre d'achat de photos en proposant pour un cliché plusieurs formats et supports (A5, A6, poster, porte-clé, ...).

18. Proposer des modifications de la base de données précédente pour qu'elle puisse prendre en charge cette nouvelle offre.