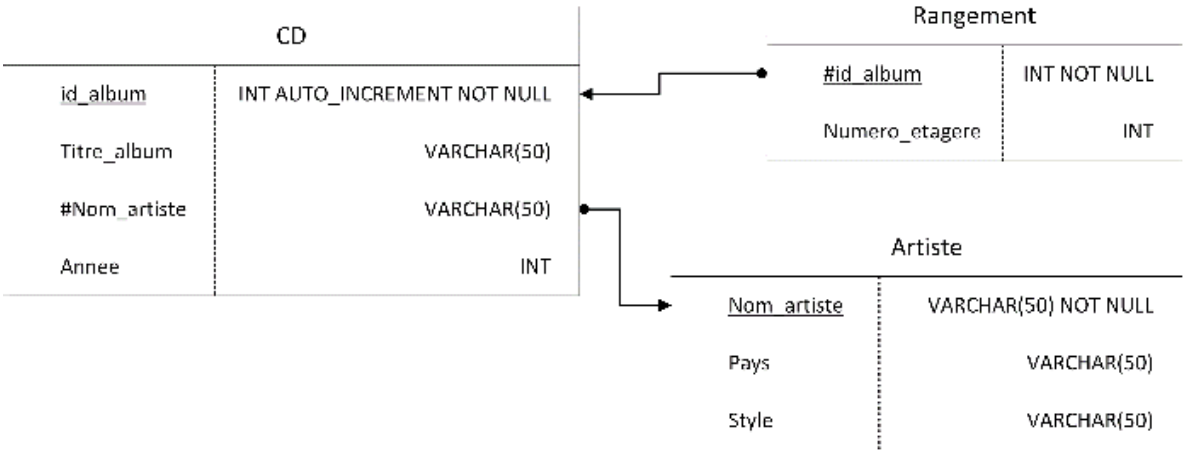
EXERCICE 1 (6 points)

Cet exercice porte sur la notion de bases de données relationnelles, le langage SQL et les protocoles de sécurisation.

Partie A – Bases de données

L'énoncé de cette partie utilise les mots du langage SQL suivants : SELECT, FROM, WHERE, JOIN, UPDATE, SET, DELETE. L'attribut AUTO_INCREMENT permet d'incrémenter automatiquement un entier dans une table à l 'insertion d'un nouvel élément.

Bob, qui dispose d'une très grande collection de CDs rangés sur plusieurs étagères numérotées, a mis en place une base de données. Voici la description des trois relations de cette base dont les clés primaires ont été soulignées et les clés étrangères indiquées par un # :



1. Indiquer, avec justification, s'il aurait été possible de choisir l'attribut Nom_artiste comme clé primaire dans la relation CD.

Dans la suite, on considère les clés étrangères suivantes :

- CD.Nom_artiste qui référence l'attribut Artiste.Nom_artiste ;
- Rangement.id_album qui référence l'attribut CD.id_album.

Voici un extrait des enregistrements des relations CD, Artiste et Rangement définies plus haut :

CD			
id_album	Titre_album	Nom_artiste	Annee
1	‘Master of Puppets’	‘Metallica’	1986
2	‘The Marshall Mathers LP’	‘Eminem’	2000
3	‘Wasting Light’	‘Foo Fighters’	2011
4	‘Wishmaster’	‘Nightwish’	2001
5	‘Dead Letters’	‘The Rasmus’	2003
6	‘Somewhere in Time’	‘Iron Maiden’	1986

Artiste		
Nom_artiste	Pays	Style
‘Nightwish’	‘Finlande’	‘Metal’
‘Foo Fighters’	‘Etats-Unis’	‘Rock’
‘Metallica’	‘Etats-Unis’	‘Metal’
‘Iron Maiden’	‘Royaume-Uni’	‘Metal’
‘Eminem’	‘Etats-Unis’	‘Rap’
‘The Rasmus’	‘Finlande’	‘Rock’

Rangement	
id_album	Numero_etagere
1	2
2	1
3	1
4	3
5	3
6	2

2. Écrire ce que renvoie la requête suivante lorsqu'on l'applique aux extraits ci-dessus.

```
SELECT Nom_artiste
FROM Artiste
WHERE Pays = "Finlande";
```

3. Écrire à présent ce que renvoie la requête suivante.

```
SELECT CD.Annee
FROM CD
JOIN Artiste
ON CD.Nom_artiste = Artiste.Nom_artiste
WHERE Artiste.Style = "Metal";
```

Bob se rend compte que l'album Wishmaster est en réalité sorti en 2000.

4. Donner la requête qu'il doit écrire pour mettre à jour sa base de données.

5. Donner la requête qu'il doit écrire pour afficher les titres de tous les albums de "Metal" rangés sur l'étagère dont le numéro est 1.

Bob a vendu l'album Dead Letters du groupe The Rasmus. Puisqu'il s'agissait du seul album de ce groupe qu'il possédait, il veut supprimer tous les enregistrements qui sont à présent inutiles dans les trois relations.

6. Donner l'ordre dans lequel il doit les supprimer en expliquant pourquoi, puis écrire la requête correspondant à la suppression de l'album dans la relation CD.

Partie B – Sécurisation

La base de données de Bob est hébergée sur un serveur auquel il accède depuis un client sur son ordinateur personnel. Pour sécuriser la connexion, un algorithme de chiffrement symétrique est utilisé.

7. Expliquer brièvement ce qu'est un algorithme de chiffrement symétrique.

La clé de chiffrement, notée C dans la suite, est choisie aléatoirement par le serveur à chaque connexion depuis un client. Afin que le chiffrement et le déchiffrement puisse se faire sans problème, le serveur doit envoyer au client la clé C de façon sécurisée.

8. Rappeler brièvement ce qu'est un algorithme de chiffrement asymétrique.

On suppose à présent que Bob possède une clé publique et une clé privée. La clé publique de Bob est supposée connue par le serveur.

9. Proposer alors une solution pour que le serveur puisse envoyer la clé C à l'ordinateur de Bob de façon sécurisée, c'est-à-dire pour que seul Bob puisse déchiffrer la clé envoyée.

EXERCICE 2 (6 points)

Cet exercice porte sur la programmation orientée objets, les tris, les algorithmes gloutons, la récursivité et les assertions.

Cet exercice est composé de trois parties dont les deux dernières sont indépendantes entre elles.

Dans cet exercice, l'entête des fonctions est décrit avec le type des objets en paramètre et le type de l'objet renvoyé. Ainsi la fonction puissance qui prend un paramètre flottant `x` et un entier `n` puis qui renvoie le flottant `x**n`, a pour entête `puissance(x: float, n: int) -> float`

Une entreprise transporte des marchandises. Elle souhaite maximiser son profit en optimisant le remplissage de ses moyens de transport. On considère qu'un moyen de transport est limité par son volume (exprimé en litres). Chaque marchandise est caractérisée par son prix (en euros) et son volume indivisible (en litres).

Supposons qu'on ait trois marchandises caractérisées par les couples (prix, volume) suivants : $m_1 = (100, 10)$, $m_2 = (100, 10)$ et $m_3 = (250, 20)$. Si le moyen de transport peut encore charger 25 litres, il vaut mieux charger la marchandise numéro 3 qui rapporte 250 € à l'entreprise plutôt que charger les marchandises numéros 2 et 3 qui rapportent 200 € au total pour le même espace utilisé.

Partie A – Quelques outils

Nous souhaitons définir une classe `Marchandise` dont chaque instance définit une marchandise possédant deux attributs entiers `prix` et `volume`.

1. Compléter le constructeur qui renvoie un objet `Marchandise`. Utiliser le mot-clé `assert` afin qu'une exception soit levée si le paramètre `v` n'est pas strictement positif.
   On rappelle que si `condition` est une expression Python booléenne s'évaluant à `True` ou `False`, l'instruction `assert condition` déclenche une exception quand la condition s'évalue à `False`.

```
class Marchandise:
    def __init__(self, p: int, v: int) -> 'Marchandise':
      ...
```

2. Donner une instruction qui permet de créer une variable `m1` représentant une marchandise d'un volume de 7 litres coûtant 20 €.

3. Proposer une méthode `ratio(self) -> float` qui renvoie le ratio prix/volume d'une marchandise.

4. Proposer une fonction `prixListe(tab: list) -> int` qui renvoie le prix cumulé de l'ensemble des marchandises formant le tableau `tab`.

Partie B – Première approche de rangement

Le transporteur souhaite maximiser son profit. On considère que nous avons les quatre marchandises définies par les couples (prix, volume) suivants :

$$m_1 = (40, 20),\ m_2 = (210, 70),\ m_3 = (160, 40) \text{ et } m_4 = (50, 50).$$

5. Préciser toutes les combinaisons de marchandises possibles si on ne dépasse pas un volume de 100 litres et le prix associé. En déduire la combinaison de marchandises qui maximise le prix.

Une première méthode appelée `ChargementGlouton` consiste à trier les marchandises dans l'ordre décroissant de leur prix volumique (ratio prix/volume), puis transporter en priorité les marchandises avec le plus grand prix volumique. Si une marchandise est trop volumineuse pour être transportée, on essaie avec la marchandise ayant le prix volumique juste inférieur, ce jusqu'à ce qu'aucune marchandise ne puisse rentrer. Ainsi, en notant `v_restant` le volume disponible et `m_i` la $(i+1)^{e}$ marchandise une fois les marchandises triées, l'algorithme peut s'écrire :

```
ChargementGlouton

n = nombre de marchandises
POUR i ALLANT de 0 à n-1 FAIRE
| SI volume de m_i <= v_restant ALORS
|   | charger m_i
|   | v_restant = v_restant - volume de m_i
TRANSPORTER le chargement prévu
```

Le tri dans l'ordre décroissant des prix volumiques donne m_3 , m_2, m_1, m_4. Si le moyen de transport accepte 100 litres de chargement, l'algorithme charge m_3 et m_1 pour un prix de 200 € (à comparer avec la combinaison trouvée précédemment pour maximiser le prix).

Par la suite, on rappelle que dans l'implémentation Python, les marchandises sont définies par des instances de la classe `Marchandise`.

On donne quelques qualificatifs : dichotomique, glouton, graphique, insertion, maximum, récursif, tri.

6. Indiquer, sans justification, le qualificatif qui s'applique le mieux à l'algorithme précédent.

7. Recopier et compléter la fonction `tri(tab: list) -> None` ci-dessous afin qu'elle trie en place un tableau contenant des objets de type `Marchandise` selon l'ordre décroissant des ratios. Ainsi, `tab[0]` doit contenir la marchandise avec le plus haut ratio prix/volume après l'appel `tri(tab)`.

```
def tri(tab: list) -> None:
    n = len(tab)
    for i in range(1, n):
        marchandise = tab[i]
        j = i-1
        while ... and ... > ... :
            tab[j+1] = ...
            j = ...
        tab[j+1] = marchandise
```

8. Sans justifier, préciser le nom de ce tri, ainsi que son coût temporel dans le pire des cas (constant, logarithmique, linéaire, quasi-linéaire ($n \log_2 n$), quadratique, cubique ou exponentiel).

9. Recopier et compléter la fonction `charge` suivante qui applique l'algorithme `ChargementGlouton` décrit plus haut.

```
def charge(tab: list, volume: int) -> list:
    tri(tab)
    chargement = []
    n = len(tab)
    for ...
        if ...
           ...
           ...
    return ...
```

Partie C – Rangement optimisé par récursivité

L'algorithme précédent ne renvoie pas toujours une solution optimale. On peut donc suivre un algorithme récursif. On note `n` le nombre de marchandises et on souhaite implémenter la fonction récursive `chargeOptimale` d'entête :

```
chargeOptimale(tab: list, v_restant: int, i: int) -> list
```

Un appel à cette fonction doit permettre de calculer la charge optimale pour un transport de volume `v_restant` utilisant les marchandises à partir de l'indice `i` :

- si `i >= n`, toutes les marchandises ont été essayées et il n'en reste plus d'autres disponibles. L'appel récursif renvoie la liste vide ;
- si `i < n` et la marchandise d'indice `i` est de volume strictement supérieur au volume restant, l'appel récursif renvoie le résultat de l'appel effectué avec le même volume restant mais avec la marchandise suivante, c'est-à-dire `chargeOptimale(tab, v_restant, i+1)` ;

- si `i < n` et la marchandise d'indice `i` est de volume inférieur ou égal au volume restant, il existe deux options possibles :
  - Option 1 soit on utilise la marchandise `i`, auquel cas le chargement contiendra cette marchandise et celles du résultat de l'appel récursif à partir de la prochaine marchandise et d'un volume restant strictement inférieur,
  - Option 2 soit on n'utilise pas la marchandise `i`, auquel cas le chargement sera le résultat de l'appel récursif avec le même volume restant mais à partir de la marchandise suivante.

  On garde l'option de chargement qui maximise le prix transporté.

10. Compléter le code de la fonction `chargeOptimale` dont le principe a été décrit ci-avant.

```
def chargeOptimale(tab: list, v_restant: int, i: int) -> list:
    if i>=...:
        return ...
    else:
        if tab[i].volume > v_restant:
            return chargeOptimale(tab, v_restant, i+1)
        else:
            option1 = chargeOptimale(tab, ..., ...)
            option2 = [tab[i]] + chargeOptimale(tab, ..., ...)
            if prixListe(option1) > prixListe(option2):
                return ...
            else:
                return ...
```

EXERCICE 3 (8 points)

Cet exercice porte sur la programmation orientée objet, les graphes et utilise la structure de données dictionnaire.

La direction de la station de ski *Le Lièvre Blanc*, spécialisée dans la pratique du ski de fond, souhaite disposer d'un logiciel lui permettant de gérer au mieux son domaine skiable. Elle confie à un développeur informatique la mission de concevoir ce logiciel. Celui-ci décide de caractériser les pistes de ski à l'aide d'une classe `Piste` et le domaine de ski par une classe `Domaine`.

Le code Python de ces deux classes est donné en **Annexe**.

Partie A – Analyse des classes `Piste` et `Domaine`

1. Lister les attributs de la classe `Piste` en précisant leur type.

La difficulté des pistes de ski de fond est représentée par 4 couleurs : verte, bleue, rouge et noire. La piste verte est considérée comme très facile, la piste bleue comme facile, la piste rouge de difficulté moyenne et la piste noire difficile. Dans la station de ski *Le Lièvre blanc*, l'équipe de direction décide de s'appuyer uniquement sur le dénivelé pour attribuer la couleur d'une piste de ski.

Ainsi, une piste de ski sera de couleur :

- `'noire'` si son dénivelé est supérieur ou égal à 100 mètres ;
- `'rouge'` si son dénivelé est strictement inférieur à 100 mètres, mais supérieur ou égal à 70 mètres ;
- `'bleue'` si son dénivelé est strictement inférieur à 70 mètres, mais supérieur ou égal à 40 mètres ;
- `'verte'` si son dénivelé est strictement inférieur à 40 mètres.

2. Écrire la méthode `set_couleur` de la classe `Piste` qui permet d'affecter à l'attribut `couleur` la chaîne de caractères correspondant à la couleur de la piste.

On exécute à présent le programme suivant afin d'attribuer la couleur adéquate à chacune des pistes du domaine skiable *Le Lièvre Blanc*.

```
1 for piste in lievre_blanc.get_pistes():
2     piste.set_couleur()
```

3. Indiquer, parmi les 4 propositions ci-dessous, le type de l'élément renvoyé par l'instruction Python `lievre_blanc.get_pistes()`.

   - Proposition A : une chaîne de caractères ;
   - Proposition B : un objet de type `Piste` ;
   - Proposition C : une liste de chaînes de caractères ;
   - Proposition D : une liste d'objets de type `Piste`.

En raison d'un manque d'enneigement, la direction de la station est souvent contrainte de fermer toutes les pistes vertes car elles sont situées généralement en bas du domaine.

4. Écrire un programme Python dont l'exécution permet de procéder à la fermeture de toutes les pistes vertes en affectant la valeur `False` à l'attribut `ouverte` des pistes concernées.

5. Écrire une fonction `pistes_de_couleur` prenant pour paramètres une chaîne de caractères `couleur` représentant la difficulté d'une piste et une liste `lst` de pistes de ski de fond. Cette fonction renvoie la liste des noms des pistes dont `couleur` est le niveau de difficulté.

   Exemple : l'instruction `pistes_de_couleur(lievre_blanc.get_pistes(), 'noire')` renvoie la liste `['Petit Bonheur', 'Forêt', 'Duvallon']`.

Un skieur de bon niveau se prépare assidûment pour le prochain semi-marathon, d'une distance de 21,1 kilomètres. À chaque entraînement, il note la liste des noms des pistes qu'il a parcourues et il souhaite disposer d'un outil lui indiquant si la distance totale parcourue est au moins égale à la distance qu'il devra parcourir le jour du semi-marathon.

La fonction `semi_marathon` donnée ci-dessous répond aux attentes du skieur : cette fonction prend en paramètre une liste `L` de noms de pistes et renvoie un booléen égal à `True` si la distance totale parcourue est strictement supérieure à 21,1 kilomètres, `False` sinon.

```
1  def semi_marathon(L):
2      distance = ...
3      liste_pistes = lievre_blanc.get_pistes()
4      for nom in L:
5          for piste in liste_pistes:
6              if piste.get_nom() == ...:
7                  distance = distance + ...
8      return ...
```

On donne ci-dessous deux exemples d'appels à cette fonction :

```
>>> entrainement1 = ['Verneys', 'Chateau enneigé', 'Rois mages',
'Diablotin']
>>> semi_marathon(entrainement1)
True
>>> entrainement2 = ['Esseillon', 'Aigle Royal', 'Duvallon']
>>> semi_marathon(entrainement2)
False
```

6. Recopier et compléter la fonction `semi_marathon`.

Partie B – Recherche par force brute

Le plan des pistes du domaine *Le Lièvre Blanc* peut être représenté par le graphe suivant :

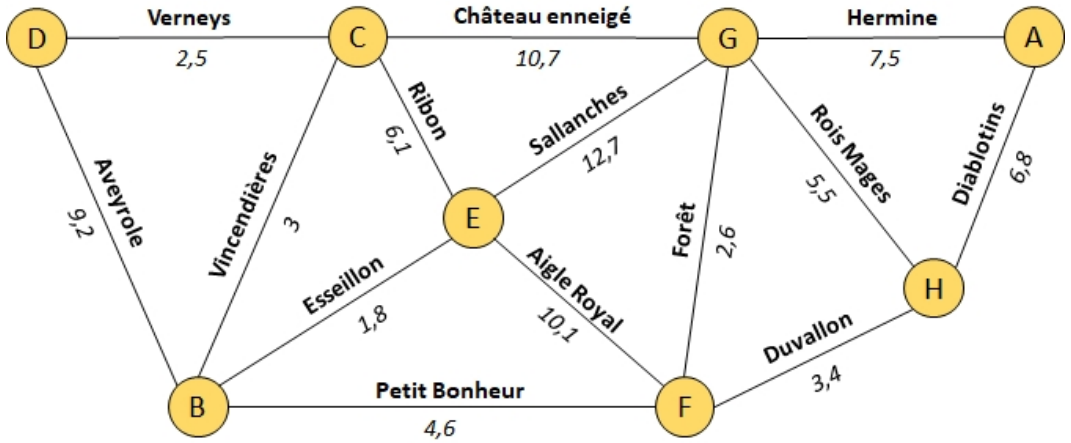


Figure 1. Graphe du domaine *Le Lièvre Blanc*

Sur chaque arête, on a indiqué le nom de la piste et sa longueur en kilomètres. Les sommets correspondent à des postes de secours.

Un pisteur-secouriste de permanence au point de secours D est appelé pour une intervention en urgence au point de secours A. La motoneige de la station étant en panne, il ne peut s'y rendre qu'en skis de fond. Il décide de minimiser la distance parcourue et cherche à savoir quel est le meilleur parcours possible. Pour l'aider à répondre à ce problème, on décide d'implémenter le graphe ci-dessus grâce au dictionnaire de dictionnaires suivant :

```
domaine = {'A' : {'G' : 7.5, 'H' : 6.8},
           'B' : {'C' : 3.0, 'D' : 9.2, 'E' : 1.8, 'F' : 4.6},
           'C' : {'B' : 3.0, 'D' : 2.5, 'E' : 6.1, 'G' : 10.7},
           'D' : {'B' : 9.2, 'C' : 2.5},
           'E' : {'B' : 1.8, 'C' : 6.1, 'F' : 10.1, 'G' : 12.7},
           'F' : {'B' : 4.6, 'E' : 10.1, 'G' : 2.6, 'H' : 3.4},
           'G' : {'A' : 7.5, 'C' : 10.7, 'E' : 12.7, 'F' : 2.6,
                  'H' : 5.5},
           'H' : {'A' : 6.8, 'F' : 3.4, 'G' : 5.5} }
```

7. Écrire une instruction Python permettant d'afficher la longueur de la piste allant du sommet `'E'` au sommet `'F'`.

8. Écrire une fonction `voisins` qui prend en paramètres un graphe `G` et un sommet `s` du graphe `G` et qui renvoie la liste des voisins du sommet `s`.

   Exemple : l'instruction `voisins(domaine, 'B')` renvoie la liste `['C', 'D', 'E', 'F']`.

9. Recopier et compléter la fonction `longueur_chemin` donnée ci-dessous : cette fonction prend en paramètres un graphe `G` et un chemin du graphe `G` sous la forme d'une liste de sommets et renvoie sa longueur en kilomètres.

   Exemple : l'instruction `longueur_chemin(domaine, ['B', 'E', 'F', 'H'])` renvoie le nombre flottant `15.3`.

```
def longueur_chemin(G, chemin):
    precedent = ...
    longueur = 0
    for i in range(1, len(chemin)):
        longueur = longueur + ...
        precedent = ...
    return ...
```

On donne ci-dessous une fonction `parcours` qui renvoie la liste de tous les chemins du graphe `G` partant du sommet `depart` et parcourant les sommets de façon unique, c'est-à-dire qu'un sommet est atteint au plus une fois dans un chemin.

Par exemple, l'appel `parcours(domaine, 'A')` renvoie la liste de tous les chemins partant du sommet A dans le graphe `domaine` sans se soucier ni de la longueur du chemin, ni du sommet d'arrivée. Ainsi, `['A', 'G', 'C']` est un chemin possible, tout comme `['A', 'G', 'C', 'B', 'E', 'F', 'H']`.

```
def parcours(G, depart, chemin = [], lst_chemins = []):
    if chemin == []:
        chemin = [depart]
    for sommet in voisins(G, depart):
        if sommet not in chemin:
            lst_chemins.append(chemin + [sommet])
            parcours(G, sommet, chemin + [sommet])
    return lst_chemins
```

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

Un appel à la fonction `parcours` précédente renvoie une liste de chemins dans laquelle figurent des doublons.

11. Recopier et compléter la fonction `parcours_dep_arr` ci-après qui renvoie la liste des chemins partant du sommet `depart` et se terminant par le sommet `arrivee` dans le graphe `G` entrés en paramètres. La liste renvoyée ne doit pas comporter de doublons. Attention, plusieurs lignes de code sont nécessaires.

```
def parcours_dep_arr(G, depart, arrivee):
    liste = parcours(G, depart)
    ...
```

12. Recopier et compléter la fonction `plus_court` donnée ci-dessous. La fonction `plus_court` prend pour paramètres un graphe `G`, un sommet de départ `depart` et un sommet d'arrivée `arrivee` ; elle renvoie un des chemins les plus courts sous la forme d'une liste de sommets.

```
def plus_court(G, depart, arrivee):
    liste_chemins = parcours_dep_arr(G, depart, arrivee)
    chemin_plus_court = ...
    minimum = longueur_chemin(G, chemin_plus_court)
    for chemin in liste_chemins:
        longueur = longueur_chemin(G, chemin)
        if ...:
            minimum = ...
            chemin_plus_court = ...
    return chemin_plus_court
```

13. Expliquer en quoi le choix fait par le pisteur-secouriste de choisir la distance minimale pour arriver le plus rapidement possible sur le lieu de l'incident est discutable. Proposer un meilleur critère de choix.

Annexe

```
1  # Pistes
2  class Piste:
3      def __init__(self, nom, denivele, longueur):
4          self.nom = nom
5          self.denivele = denivele   # en mètres
6          self.longueur = longueur   # en kilomètres
7          self.couleur = ''
8          self.ouverte = True

9      def get_nom(self):
10         return self.nom

11     def get_longueur(self):
12         return self.longueur

13     def set_couleur(self):
14         # À compléter

15     def get_couleur(self):
16         return self.couleur

17 # Domaine skiable
18 class Domaine:
19     def __init__(self, a):
20         self.nom = a
21         self.pistes = []

22     def ajouter_piste(self, nom_piste, denivele, longueur):
23         self.pistes.append(Piste(nom_piste, denivele, longueur))

24     def get_pistes(self):
25         return self.pistes

26 # Programme principal
27 lievre_blanc = Domaine("Le Lièvre Blanc")
28 lievre_blanc.ajouter_piste('Aveyrole', 62, 9.2)
29 lievre_blanc.ajouter_piste('Verneys', 10, 2.5)
30 lievre_blanc.ajouter_piste('Vincendières', 45, 3)
31 lievre_blanc.ajouter_piste('Ribon', 70, 6.1)
32 lievre_blanc.ajouter_piste('Esseillon', 8, 1.8)
33 lievre_blanc.ajouter_piste('Petit Bonheur', 310, 4.6)
34 lievre_blanc.ajouter_piste('Aigle Royal', 85, 10.1)
35 lievre_blanc.ajouter_piste('Château enneigé', 54, 10.7)
36 lievre_blanc.ajouter_piste('Sallanches', 78, 12.7)
37 lievre_blanc.ajouter_piste('Forêt', 145, 2.6)
```

```
lievre_blanc.ajouter_piste('Hermine', 27, 7.5)
lievre_blanc.ajouter_piste('Rois mages', 42, 5.5)
lievre_blanc.ajouter_piste('Diablotin', 76, 6.8)
lievre_blanc.ajouter_piste('Duvallon', 200, 3.4)
```

EXERCICE 1 (6 points)

Cet exercice porte sur la programmation en Python en général, la programmation orientée objet et la récursivité.

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 :

```
S * *
    * *
      *
      * * * * *
              E
```

- `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 :

```
1  class Chemin:
2
3      def __init__(self, itineraire):
4          self.itineraire = itineraire
5          longueur, largeur = 0, 0
6          for direction in self.itineraire:
7              if direction == "D":
8                  longueur = longueur + 1
9              if direction == "B":
10                 largeur = largeur +1
11         self.longueur = longueur
12         self.largeur = largeur
13         self.grille = [['.' for i in range(longueur+1)] for j in
                          range(largeur+1)]
14
15     def remplir_grille(self):
16         i, j = 0, 0               # Position initiale
17         self.grille[0][0] = 'S'   # Case de départ marquée d'un S
18         for direction in ...:
19             if direction == 'D':
20                 ... = ...       # Déplacement vers la droite
21             elif direction == 'B':
22                 ... = ...       # Déplacement vers le bas
23             self.grille[i][j] = '*'  # Marquer le chemin avec '*'
24         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.largeur
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. Écrire 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. Écrire 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 `*` 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. Écrire 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 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 $m \times n$.