Centres étrangers - juin 2025 - sujet 1

Exercice 1 (Graphes et POO - 6 points)
Certains sportifs sont adeptes des courses d'orientation : munis d'une fiche de contrôle, d'une carte et d'une boussole, les participants doivent trouver leur itinéraire dans le but de relier des points de contrôle, appelés balises. Chaque balise est numérotée et équipée d'un poinçon permettant de l'identifier : lorsqu'un participant trouve une balise, il poinçonne sa fiche de contrôle avec ce poinçon, différent d'une balise à l'autre.

Le site *La forêt des chênes* propose trois itinéraires de courses d'orientation, de niveaux différents : un itinéraire vert de niveau facile, un itinéraire rouge de niveau moyen, et un itinéraire noir de niveau difficile.

Partie A
Le site *La forêt des chênes* est représenté par le graphe ci-dessous dans lequel chaque nœud correspond à une balise et chaque arête représente un chemin reliant deux balises. Pour faire référence aux couleurs des itinéraires, on utilise des symboles avec la correspondance suivante :

- un triangle pour indiquer que l'itinéraire vert passe par cette balise ;
- un carré pour indiquer que l'itinéraire rouge passe par cette balise ;
- un pentagone pour indiquer que l'itinéraire noir passe par cette balise.

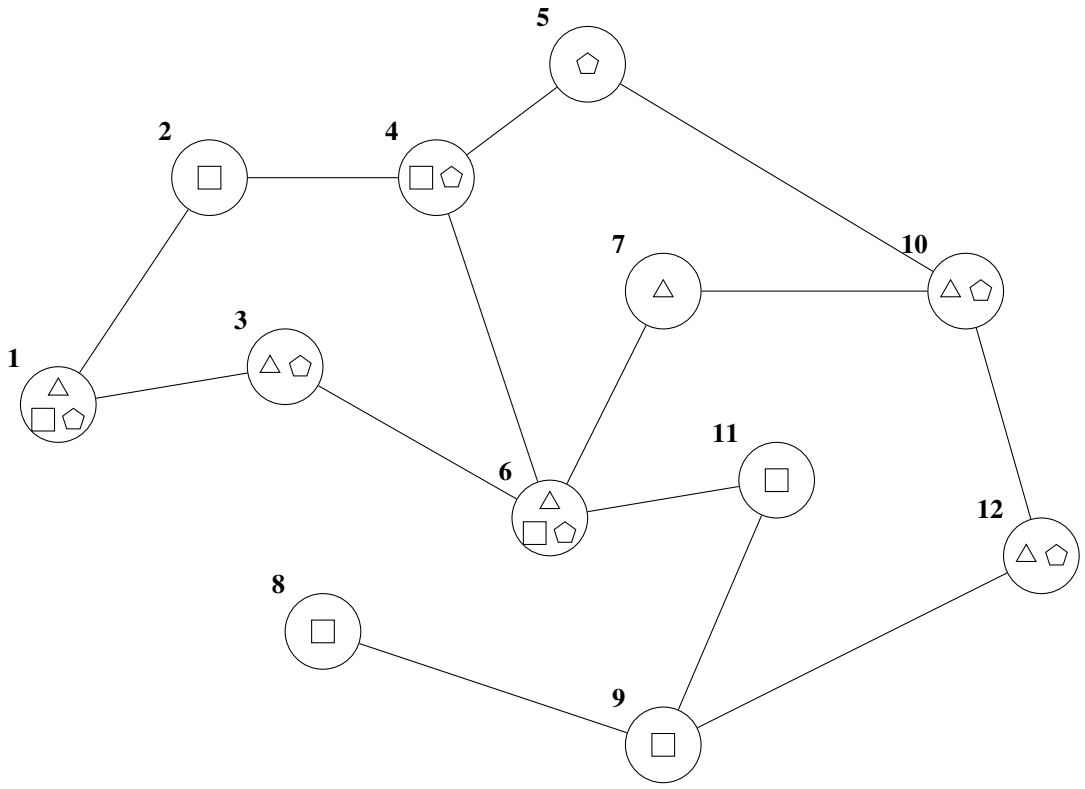


FIGURE 1 – Graphe représentant le site *La forêt des chênes*

Une balise peut donc faire partie d'un ou de plusieurs itinéraires.

Le point de départ est commun aux trois itinéraires : il s'agit de la balise portant le numéro 1. On peut alors constater, par lecture du graphe, que l'itinéraire vert et l'itinéraire noir se terminent à la balise numéro 12, tandis que l'itinéraire rouge se termine à la balise numéro 8.

On choisit de modéliser ce site en Python en utilisant une classe `Balise` dont le code est donné en annexe. Cette annexe n'est pas à rendre avec la copie.

Chaque balise du site est représentée par une instance de la classe `Balise` et a pour attributs :

- `num_balise`, un nombre entier correspondant au numéro de la balise ;
- `couleurs_balise`, une liste de chaîne de caractères dont chaque élément est une couleur de la balise ;
- `voisines`, une liste d'instances de la classe `Balise` ;
- `visitee`, un indicateur booléen qui indique si la balise a déjà été poinçonnée ou non (cet attribut utilisé dans la partie B).

1. Recopier et compléter le code de la ligne 28 afin d'instancier la balise portant le numéro 12.
2. Ecrire l'instruction de la ligne 38 permettant de compléter l'implémentation du graphe de la Figure 1.
3. Dans la console, on saisit l'instruction suivante :

```
>>> balise4.methode1()
```

   Indiquer le résultat renvoyé à l'écran.
4. Dans la console, on saisit la suite d'instructions suivante :

```
>>> balise4.methode2('rouge')
>>> balise4.methode3('vert')
>>> balise4.couleurs_balise
```

   Indiquer le résultat renvoyé à l'écran.

On instancie à nouveau la balise numéro 4 pour revenir au graphe représenté sur la Figure 1.

Ci-après, on donne le code incomplet d'une fonction `itineraire`. Cette fonction renvoie, sous la forme d'une liste, les numéros des balises qui correspondent à l'itinéraire reliant `balise_debut` et `balise_fin`, et dont le niveau de difficulté (vert, rouge ou noir) est donné par le paramètre `couleur`. Par exemple, l'appel `itineraire(balise1, balise12, 'vert')` renvoie la liste d'entiers `[1, 3, 6, 7, 10, 12]`.

```
1  def itineraire(balise_debut, balise_fin, couleur):
2      assert couleur in balise_debut.couleurs_balise
3      assert couleur in balise_fin.couleurs_balise
4      balise = balise_debut
5      chemin = [balise]
6      while balise.num_balise != ... :
7          for b in balise.voisines:
8              if (couleur in ...) and (b not in ...):
9                  balise =...
10                 chemin.append(balise)
11     return [b.num_balise for b in chemin]
```

5. Recopier et compléter les lignes 4 à 8 de la fonction `itineraire`.

Le site *La forêt des chênes* décide de définir une nouvelle épreuve en récompensant les participants qui reviennent avec l'ensemble des balises poinçonnées sur leur fiche de contrôle. Les couleurs des balises ne sont donc pas prises en compte dans cette épreuve.

On rappelle que le point de départ de tous les itinéraires est la balise portant le numéro 1. Un participant décide ainsi, pour remplir cet objectif, d'appliquer un algorithme de parcours en profondeur du graphe, en choisissant la balise de plus petit numéro lorsqu'il y a plusieurs choix possibles.

6. Donner, dans l'ordre, les numéros des balises rencontrées par ce participant.

Partie B

Dans cette partie, les couleurs des balises ne sont plus prises en compte.

Le site *La forêt des chênes* est également recommandé par les amateurs de trails (courses à pied en milieu naturel) à la recherche de performances. Ainsi, sur chaque arête du graphe ci-dessous, on mentionne le temps moyen, en minute, permettant de relier deux balises voisines, les balises étant reliées par des chemins plats.

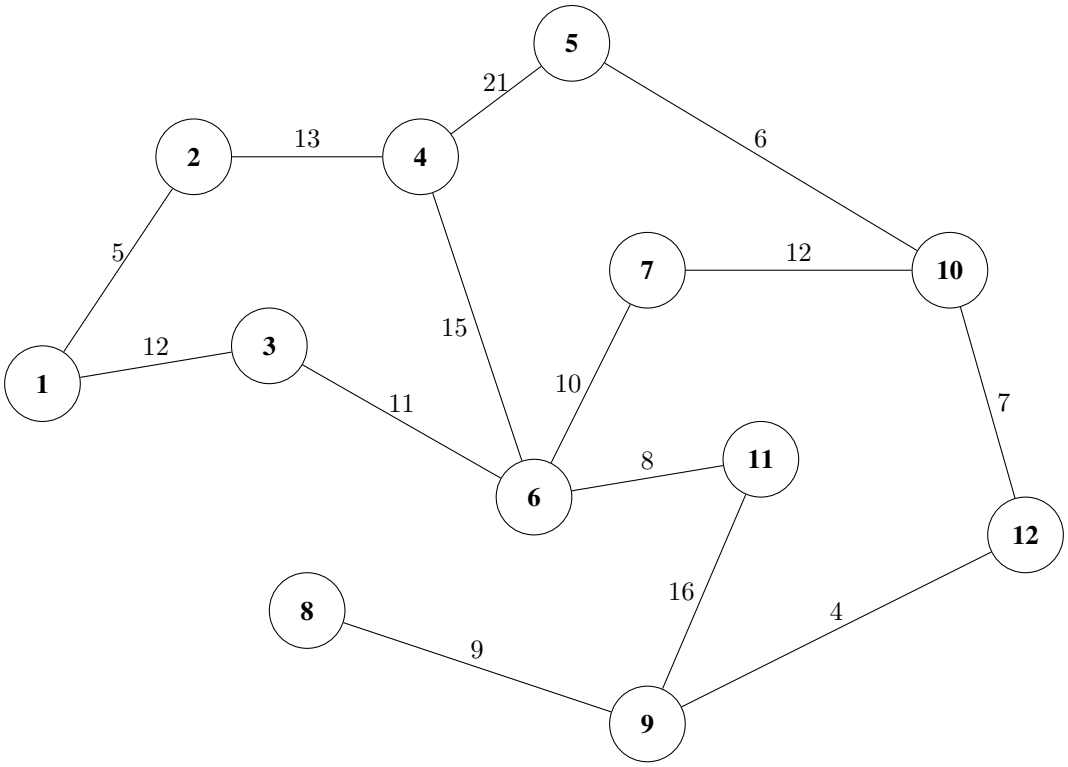


FIGURE 2 – Graphe représentant le site *La forêt des chênes* avec indication des temps moyens en minute

On modifie alors l'implémentation du graphe en Python en remplaçant la liste des balises voisines par une liste de tuples indiquant la balise voisine et le temps moyen pour la rejoindre. Par exemple, le code de la ligne 30 concernant la balise 1 est remplacé par : `balise1.voisines = [(balise2, 5), (balise3, 12)]`.

7. Donner le code qui permet, de la même façon, de définir les balises voisines de la balise portant le numéro 4 et qui permet donc de remplacer l'instruction de la ligne 33 dans le code donné en annexe.

On donne ci-dessous le code d'une fonction Python `mystere` prenant pour paramètre `balise`, une instance de la classe `Balise`.

```
1 def mystere(balise):
2     meilleure_balise = None
3     mini = -1
4     for b, t in balise.voisines:
5         if (b.visitee == False) and (mini == -1 or t < mini):
6             meilleure_balise, mini = b, t
7     return meilleure_balise
```

Dans la question suivante, on suppose que la valeur de l'attribut `visitee` de toutes les instances de la classe `Balise` est `False`.

8. Indiquer le résultat renvoyé à l'écran lorsque l'utilisateur saisit, dans la console, l'instruction suivante :

```
>>> mystere(balise10).num_balise
```

Dans la suite de l'exercice, on s'intéresse à un sportif qui, pour déterminer son itinéraire, choisit, à chaque balise, de s'orienter vers la balise voisine la plus proche en temps, sans jamais repasser par une balise déjà visitée. Ce sportif part de la balise numéro 1 pour rejoindre la balise numéro 12.

9. Donner, dans l'ordre, les numéros des balises rencontrées par ce sportif.

On donne ci-dessous le script, incomplet, d'une fonction `itineraire_trail` prenant pour paramètres `balise_debut` et `balise_fin`, deux instances de la classe `Balise`. Cette fonction doit renvoyer, sous la forme d'une liste d'entiers, les numéros des balises rencontrées successivement par le sportif lorsqu'il part de `balise_debut` pour rejoindre `balise_fin`. Si un tel itinéraire ne peut pas être déterminé, la fonction doit renvoyer `None`.

On supposera qu'avant chaque appel à la fonction `itineraire_trail`, l'utilisateur aura exécuté les instructions permettant d'attribuer la valeur `False` à l'attribut `visitee` de toutes les instances de la classe `Balise`.

```
1  def itineraire_trail(balise_debut, balise_fin):
2      balise_debut.visitee = True
3      balise = balise_debut
4      chemin = [balise]
5      while balise_fin not in chemin:
6          prochaine = ...
7          if prochaine != None:
8              ...
9          else:
10             return None
11     return [b.num_balise for b in chemin]
```

10. Recopier les lignes 5 à 9 en complétant la ligne 6 et en ajoutant autant de lignes que nécessaires à la place de la ligne 8 afin de compléter la fonction `itineraire_trail`.
11. Indiquer, sans justifier, à quel type d'algorithme appartient le script précédent parmi les trois propositions suivantes :
    - **Proposition A :** algorithme glouton ;
    - **Proposition B :** algorithme de recherche par force brute ;
    - **Proposition C :** algorithme des k plus proches voisins.
12. Donner un avantage et un inconvénient de ce type d'algorithme.

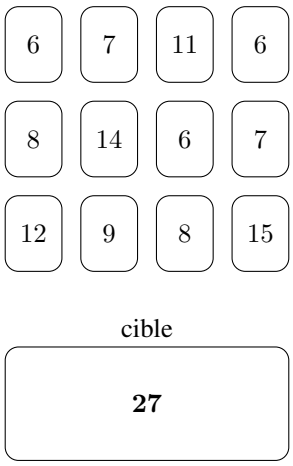
```
class Balise:
    def __init__(self, numero, couleurs):
        self.num_balise = numero
        self.couleurs_balise = couleurs
        self.voisines = []
        self.visitee = False

    def methode1(self):
        return [b.num_balise for b in self.voisines]

    def methode2(self, couleur):
        self.couleurs_balise = [c for c in self.couleurs_balise if c != couleur]

    def methode3(self, couleur):
        self.couleurs_balise.append(couleur)

balise1 = Balise(1, ['vert', 'rouge', 'noir'])
balise2 = Balise(2, ['rouge'])
balise3 = Balise(3, ['vert', 'noir'])
balise4 = Balise(4, ['rouge', 'noir'])
balise5 = Balise(5, ['noir'])
balise6 = Balise(6, ['vert', 'rouge', 'noir'])
balise7 = Balise(7, ['vert'])
balise8 = Balise(8, ['rouge'])
balise9 = Balise(9, ['rouge'])
balise10 = Balise(10, ['vert', 'noir'])
balise11 = Balise(11, ['rouge'])
balise12 = ...

balise1.voisines = [balise2, balise3]
balise2.voisines = [balise1, balise4]
balise3.voisines = [balise1, balise6]
balise4.voisines = [balise2, balise5, balise6]
balise5.voisines = [balise4, balise10]
balise6.voisines = [balise3, balise4, balise7, balise11]
balise7.voisines = [balise6, balise10]
balise8.voisines = [balise9]
...
balise10.voisines = [balise5, balise7, balise12]
balise11.voisines = [balise6, balise9]
balise12.voisines = [balise9, balise10]
```

Exercice 2 (Algorithmique, récursivité et files - 6 points)
On cherche à implémenter en Python le jeu de cartes **soupe de nombres** créé par *Javier Dominguez Cruz* et paru aux éditions *EDGE*.

Il s'agit d'un jeu de calcul mental où les joueurs doivent réaliser un nombre à l'aide d'une succession d'opérations mathématiques élémentaires : addition, soustraction, multiplication et division. Par soucis de simplification, dans tout cet exercice, la seule opération utilisée sera **l'addition**.

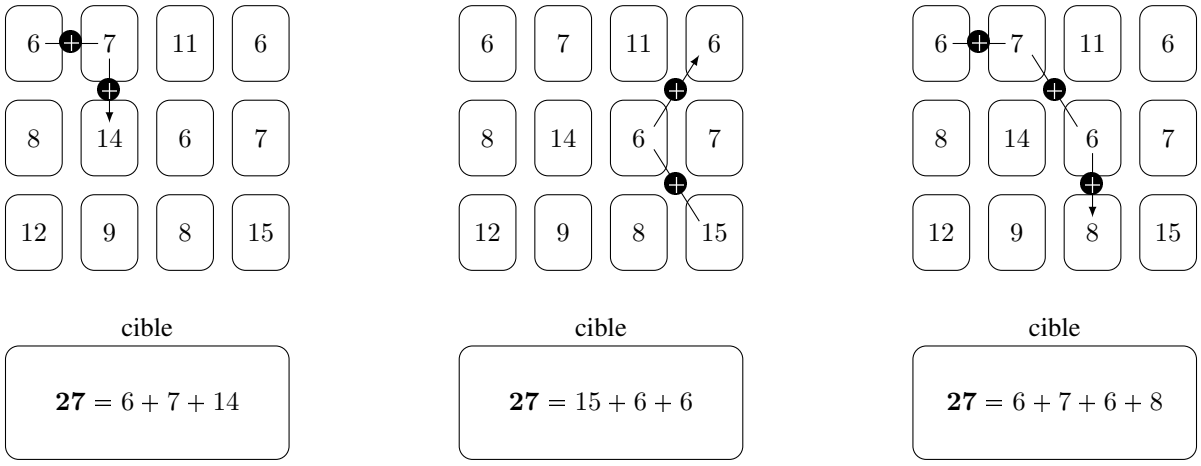
Tout d'abord, on dispose d'un jeu de cartes sur lesquelles est inscrit un nombre. On sélectionne aléatoirement 12 cartes que l'on dispose sous la forme d'une grille de 3 lignes et 4 colonnes. Dans la suite, une telle grille sera appelé *plateau de jeu*. Ensuite, plusieurs manches sont jouées en choisissant un nombre cible aléatoire.

Un exemple de manche est présenté dans le schéma suivant.



L'objectif est de trouver une plus courte chaîne valide (celle dont le nombre de cartes est le plus petit possible) dont la somme vaut la cible (cette chaine peut être constituée d'un seul nombre). Pour constituer une chaine, deux cartes consécutives de cette chaîne doivent être voisines dans la grille, horizontalement, verticalement ou diagonalement. Une même carte ne peut pas être utilisée plusieurs fois dans une chaîne.

Avec le plateau ci-dessus, on peut obtenir le nombre 27 de plusieurs manières. Le schéma suivant présente trois exemples de chaînes pour obtenir 27. Les deux premières, de longueur 3, sont les plus courtes possibles.



On remarque que les enchainements suivants ne sont pas des chaînes :

- $12 + 15$ car les cartes ne sont pas voisines ;
- $9 + 9 + 8$ car la carte contenant le nombre 9 est utilisée plusieurs fois.

Afin d'implémenter ce jeu en Python, on représente le plateau de nombres par la liste de ses lignes, chacune d'entre elles étant représentée par une liste d'entiers. Ainsi, le plateau précédent sera représenté par :

```
plateau = [[6, 7, 11, 6],
           [8, 14, 6, 7],
           [12, 9, 8, 15]]
```

Le but est de proposer un code qui simule le jeu et propose une des meilleures solutions. On va considérer une extension des règles précédentes où les plateaux sont de taille quelconque.
Soit le code Python suivant :

```
v = plateau[1][2]
```

1. Donner la valeur de `v` obtenue.

On souhaite écrire une fonction `plateau_init` telle que `plateau_init(n, m, cartes)` où :

- `n` et `m` sont deux entiers naturels non nuls et
- `cartes` est une liste d'au moins `n` × `m` entiers

renvoie un plateau à `n` lignes et `m` colonnes comportant des nombres choisis au hasard parmi `cartes` sans remise.

La fonction `shuffle` du module `random` mélange aléatoirement une liste en la modifiant en place.

2. Compléter le programme suivant :

```
1 def plateau_init(n, m, cartes):
2     shuffle(cartes)
3     plateau = ...
4     for i in range(...):
5         plateau.append([cartes[i+j*n] for j in range(m)])
6     return ...
```

3. Compléter la fonction `cartes_voisines` telle que `carte_voisines(n, m, i, j)`, où `(i, j)` sont les coordonnées d'une carte dans un plateau à `n` lignes et `m` colonnes, renvoie la liste des coordonnées des cartes voisines de `(i, j)`.

```
1 def cartes_voisines(n, m, i, j):
2     voisines = []
3     for i2 in range(i-1, i+2):
4         for j2 in ...:
5             if (i2, j2) != (i, j) and i2 in range(n) and ...:
6                 voisines.append( ... )
7     return ...
```

On va représenter une chaîne comme une liste de coordonnées des cartes correspondantes. Ainsi, la chaîne $6 + 7 + 6 + 8$ présente dans le schéma précédent sera représentée par la liste Python suivante :

```
chaine = [(0,0), (0,1), (1,2), (2,2)]
```

Soit l'enchainement suivant :

```
e1 = [(2,1), (1,0), (0,0), (0,1)]
e2 = [(1,1), (1,2), (2,0)]
e3 = [(0,2), (0,3), (1,4)]
```

4. Donner la valeur associée à `e1`, `e2` et `e3` (si la chaîne existe)
5. Ecrire une fonction `chaine_evalue` qui prend en paramètres un plateau et une chaîne, et renvoie la valeur associée à une telle chaîne. Par exemple, avec la chaîne précédente `chaine_evalue(plateau, chaine)` renverra l'entier `27`.

Pour réaliser l'exploration et obtenir une chaîne la plus courte possible, on va considérer un parcours en largeur depuis chaque carte du plateau.

6. Expliquer pourquoi il est pertinent d'utiliser un parcours en largeur plutôt qu'un parcours en profondeur pour réaliser l'exploration permettant d'obtenir une chaine la plus courte possible.

Afin de réaliser ce parcours, on suppose définie une structure de donnée `file` munie des quatre opérations suivantes :

- `file_init` telle que `file_init()` renvoie une nouvelle file vide ;
- `file_ajoute` telle que `file_ajoute(f, x)` ajoute `x` dans la file `f` ;
- `file_retire` telle que `file_retire(f)` retire l'élément de `f` le plus anciennement ajouté, si la file est vide l'exception `ValueError` sera émise ;
- `file_est_vide` telle que `file_est_vide(f)` renvoie un booléen indiquant si la file `f` est vide.

7. Recopier et compléter les lignes incomplètes de la fonction `explore` qui réalise le parcours.

```
def explore(plateau, cible):
    n, m = len(plateau), len(plateau[0])
    a_visiter = ...
    for i in range(n):
        for j in range(m):
            file_ajoute(a_visiter, [(i,j)])
    while ...:
        chemin = file_retire(a_visiter)
        if chaine_evalue(plateau, chemin) == ...:
            return chemin
        dernier = chemin[-1]
        i0, j0 = dernier
        for i, j in cartes_voisines(n, m, i0, j0):
            if (i, j) not in ...:
                file_ajoute(a_visiter, chemin + [(i,j)])
    # Pas de solutions
    return None
```

8. Expliquer, sans écrire de code mais en décrivant votre solution, comment modifier la fonction `explore` afin qu'elle renvoie la liste de tous les chemins solutions.

Exercice 3 (Listes, dictionnaires, POO, arbres binaires de recherche et bases de données - 8 points)

Un ballon-sonde (utilisé en météorologie et en astronautique) est un ballon à gaz libre utilisé pour faire des mesures locales dans l'atmosphère. Chaque mesure réalisée est horodatée et géolocalisée par GPS.

À la fin de leur vol, les radiosondes redescendent vers la terre, se posent « n'importe où » et sont « abandonnées » à leur sort. C'est pourquoi vous pouvez en trouver. Diane Delstar est une radio amatrice appartenant à un club de passionnés. Le club récupère les ballons équipés de leur radiosonde et répertorie toutes les infos de leur atterrissage sur la terre ferme dans une base de données consultable sur un site web (source : https ://sondehub.org). Le programme, écrit par Diane, lui permet de récupérer les données relevées par les sondes. La variable enregistrement fait référence à un dictionnaire qui contient un ensemble de données pour une sonde.

Exemple :

```
enregistrement = {'num_serie': 623, 'altitude': 1150.0,
                  'datetime': '2024-06-27T23:36:01.000000Z',
                  'latitude': 38.38825, 'longitude': 27.09004}
```

1. Ecrire la ligne de code qui, dans la console, permet d'afficher la valeur `38.38825` de la variable `enregistrement`.

Les valeurs dont la clé est `'datetime'` sont des chaînes de caractères.

Exemple : `'2024-06-27T23:36:01.000000Z'` est une chaîne de caractères qui comprend la date au format année-mois-jour (`'2024-06-27'`), suivie de l'indicatif (`'T'`), puis de l'heure (`'23:36:01.000000'`) et enfin d'un indicatif de fuseau horaire (`'Z'`).

Diane écrit la fonction `nettoyage_datetime` qui prend en paramètre une chaîne de caractères correspondant à la clé `'datetime'` au format ISO 8601.

```
1 def nettoyage_datetime(chaine):
2     date=''
3     horaire=''
4     for i in range(10):
5         date=date+chaine[i]
6     for i in range(8):
7         horaire=horaire+chaine[i+11]
8     return date,horaire
```

2. Ecrire et expliquer ce que renvoie l'appel de la fonction `nettoyage_datetime ('2024-06-27T23:36:01.000000Z')`

Tous les enregistrements de données de plusieurs radiosondes sont stockés dans la variable `frames` de type list dont un extrait est donné ci-dessous :

```
frames = [{'num_serie': 623, 'altitude': 620.1,
           'datetime': '2024-07-01T01:58:55.000Z',
           'latitude': 43.20223, 'longitude': -72.05708},
          {'num_serie': 500, 'altitude': 6375.75,
           'datetime': '2024-07-01T23:35:32.000Z',
           'latitude': -20.8759, 'longitude': 55.58805},
          {'num_serie': 623, 'altitude': 622.6,
           'datetime': '2024-07-01T01:59:01.000Z',
           'latitude': 43.20224, 'longitude': -72.05711},
          {'num_serie': 700, 'altitude': 60000.0,
           'datetime': '2024-07-01T11:51:23.000000Z',
           'latitude': 40.87873, 'longitude': 29.09587}]
```

3. Donner les valeurs renvoyées par les appels `len(frames)` et `len(frames[1])`.

Une radiosonde peut monter au maximum à une altitude de 35 000 mètres et, bien sûr, ne peut atteindre une altitude de valeur négative. La fonction `detecter_anomalie` prend en paramètre un élément d'une liste analogue à `frames` et renvoie un booléen indiquant si la clé `'altitude'` est associée à une valeur négative ou à une valeur strictement supérieure à 35 000 mètres.

Exemple :

```
>>> detecter_anomalie({'num_serie': 700, 'altitude': 60000.0,
'datetime': '2024-07-01T11:51:23.000000Z',
'latitude': 40.87873, 'longitude': 29.09587})
True
```

4. Ecrire, en langage Python, la fonction `detecter_anomalie`.

Diane a besoin d'obtenir la liste des numéros de séries des radiosondes. Elle écrit donc la fonction `liste_num_serie`.

Une sonde ne doit figurer qu'une seule fois dans la liste représentée par son numéro de série. Exemple d'appel de la fonction `liste_num_seri` avec comme paramètre la variable `frames` ci-dessus :

```
>>> liste_num_serie(frames)
[623, 500, 700]
```

On rappelle que l'opérateur `in` permet de réaliser un test d'appartenance. Par exemple :

```
>>> fruits=['Banane','Pomme','Poire']
>>> 'Banane' in fruits
True
>>> 'Orange' in fruits
False
```

5. Ecrire, en langage Python, la fonction `liste_num_serie`.

En exploitant les enregistrements de données d'une même sonde, Diane décide d'en déterminer la distance totale parcourue. La fonction `distance_haversine`

- prend en paramètre deux tuples de coordonnées `(latitude, longitude)` ;
- renvoie la distance, en kilomètres, entre les deux points.

Exemple : Les points `point1` et `point2` ont les coordonnées géographiques :

- Point 1 : `latitude1 = 46.815` et `longitude1 = 6.943`
- Point 2 : `latitude2 = 47.049` et `longitude2 = 7.52828`

```
>>> point1 = (46.815,6.943)
>>> point2 = (47.049,7.52828)
>>> distance_haversine(point1, point2)
51.5
```

La fonction `distance_totale` prend en paramètre une liste de tuples de coordonnées (formées d'une latitude et d'une longitude) constituant le déplacement de la sonde. Elle additionne les distances entre les points successifs et renvoie la distance totale parcourue. Exemples :

```
>>> deplacement = [(46.8125, 6.9433), (46.91498, 7.23039), (47.00661, 7.48054)]
>>> distance_totale(deplacement)
46.1
>>> deplacement = [(46.8125, 6.9433), (46.91498, 7.23039), (47.00661, 7.48054),
(47.2512,7.5201)]
>>> distance_totale(deplacement)
73.5
```

6. Compléter la fonction `distance_totale` :

```
def distance_totale(dep):
    total = 0
    for i in range(...):
        ...
    return total
```

Lorsque les sondes atterrissent à la fin de leur voyage, elles sont récupérées par les membres du club. Les dernières positions des sondes sont ensuite enregistrées pour être affichées sur le site Web du club. Un arbre binaire de recherche sera exploité par Diane pour gérer la récupération des sondes :

- chaque sonde est associée à un arbre binaire de recherche (ou plus précisément à sa racine) ;
- les arbres binaires de recherche sont des instances de la classe `Sonde` ;
- les instances de la classe `Sonde` possèdent les attributs `num_serie`, `latitude`, `longitude` et `date` qui décrivent la sonde ainsi que des attributs `gauche` et `droite` qui font références aux deux sous-arbres reliés à la racine ;
- l'arbre binaire de recherche vide est une instance de `Sonde` dont tous les attributs valent `None` ;
- la méthode `est_vide` renvoie un booléen qui permet de savoir si l'arbre représenté est vide ;
- si le sous-arbre gauche n'est pas vide, alors son attribut `num_serie` est strictement inférieur à l'attribut `num_serie` de la racine ;
- si le sous-arbre droit n'est pas vide, alors son attribut `num_serie` est strictement supérieur à l'attribut `num_serie` de la racine.

Les premières lignes de code de la classe `Sonde` sont les suivantes :

```
class Sonde:
    def __init__(self, num_serie, latitude, longitude, date, gauche, droit):
        self.num_serie = num_serie
        self.latitude = latitude
        self.longitude = longitude
        self.date = date
        self.gauche = gauche
        self.droit = droit

    def est_vide(self):
        return self.num_serie is None
```

7. Ecrire la ligne de code qui permet de créer une instance `sonde623` de la classe `Sonde` à l'aide des paramètres suivants : `num : 623; lati : 38.38825; long : 27.09004; date : '2024-06-27'`

Ci-dessous, un extrait de la représentation graphique de l'arbre binaire de recherche. Les numéros de série (`num_serie`) de chaque sonde y sont indiqués.

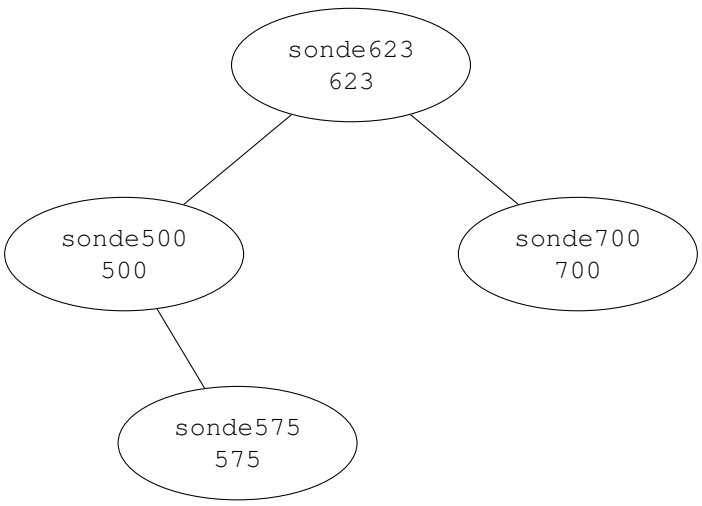


FIGURE 1 – Arbre binaire de recherche

8. Recopier et compléter la représentation graphique, ci-dessus, en y ajoutant successivement et dans cet ordre les nœuds suivants :
   - ⋆ sonde portant le nom `sonde900` et le numéro de série `900` ;
   - ⋆ sonde portant le nom `sonde650` et le numéro de série `650` ;
   - ⋆ sonde portant le nom `sonde300` et le numéro de série `300`.
9. L'arbre peut être parcouru selon l'ordre de parcours *préfixe*, *infixe*, *suffixe* ou *en largeur d'abord*. Indiquer celui qui permet de lire les numéros des sondes dans l'ordre croissant des numéros de série.
10. Compléter la méthode `rechercher` (lignes 1, 3, 4 et 7), ci-dessous, qui prend en paramètre un numéro de série (`numero`) et qui renvoie l'instance de la classe `Sonde` correspondant au numéro de série ou `None` si la sonde n'a pas été référencée dans l'arbre.

```
1  def rechercher(..., numero):
2      if self.est_vide():
3          return ...
4      if numero == ...:
5          return self
6      elif numero < self.num_serie:
7          return self.....rechercher(numero)
8      else:
9          return self.droit.rechercher(numero)
```

11. Expliquer pourquoi la méthode `rechercher` est une méthode récursive.

Finalement, Diane, après avoir traité toutes les données issues des enregistrements des sondes, décide pour son site web d'exploiter une base de données SQL. Son objectif est de référencer toutes les radiosondes récupérées par les membres du club. La base de données simplifiée a pour nom : `InventaireSondesRetrouvees`

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` et `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`.

`InventaireSondesRetrouvees` est composée de trois relations dont des extraits sont donnés ci-dessous :

abonne			
id_abonne	nom	prenom	email
32	Détoile	Diane	D.Detoile@nsi.fr
15	Petit	Claire	P.Claire@nsi.fr
24	Girard	Antoine	A.Girard@nsi.fr
16	Lemoine	Martin	M.Lemoine@huit.fr

sonde			
num_serie	modele	constructeur	date_lancement
623	RS41	Vaisala	2024-07-10
500	M10	Météomodem	2024-07-05
900	M20	Météomodem	2024-07-17
480	RS41	Vaisala	2024-06-20
810	WxR	Weathex	2024-06-05

info_recuperation					
ref	num_serie	id_abonne	date_recup	latitude	longitude
10	900	15	2024-08-05	47.999	2.549
11	500	24	2024-07-06	47.159	3.151
12	623	15	2024-07-18	47.257	11.974
13	810	32	2024-06-10	44.374	22.448

12. Citer, en justifiant votre réponse, un attribut pouvant servir de clé primaire dans la relation `abonne`.
13. Citer les attributs qui sont des clés étrangères dans la relation `infos_recuperation`.
14. Ecrire l'affichage que provoque l'exécution de la requête SQL ci-dessous sur les extraits de la base de données `InventaireSondesRet`

```
SELECT nom,prenom FROM Abonnes WHERE id_abonne>20
```

La sonde de numéro de série 480 a été retrouvée par Antoine Girard le 2024-07-10 à la latitude 47.230 et la longitude 12.244.

15. Ecrire la requête SQL qui permet d'ajouter les données de récupération de la sonde 480 dans la relation `nfos_recuperation`. L'attribut `ref` doit prendre la valeur `14`.

On souhaite afficher plusieurs renseignements sur les radiosondes récupérées.

16. Ecrire la requête SQL qui pour chaque sonde affiche le numéro de série, le nom de l'abonné, et la date de récupération.