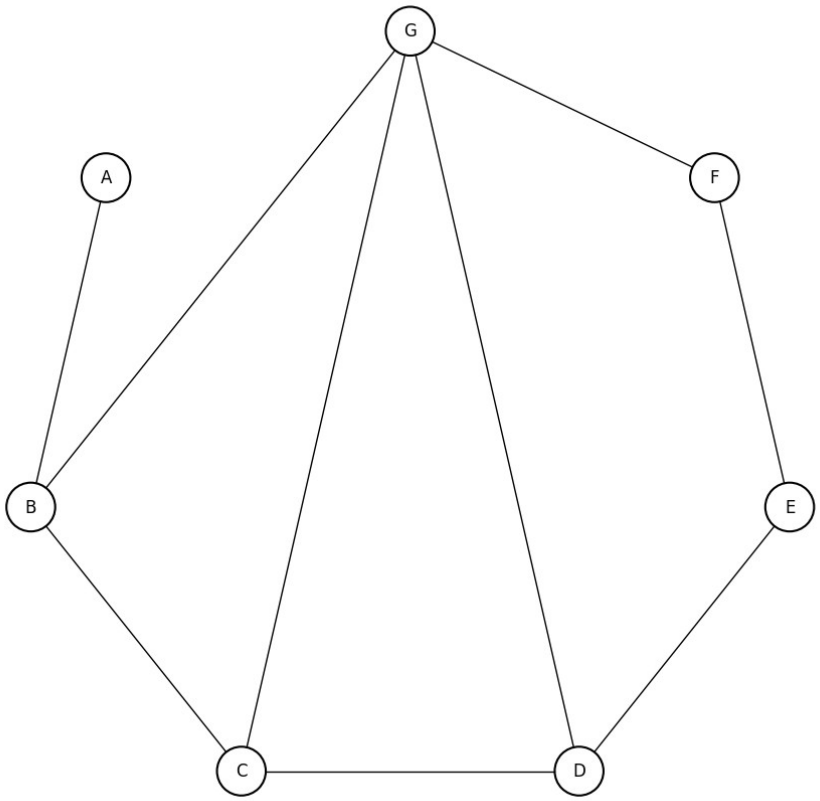
Exercices de révision

Exercice 1

Soit le graphe non-valué et non-orienté suivant:



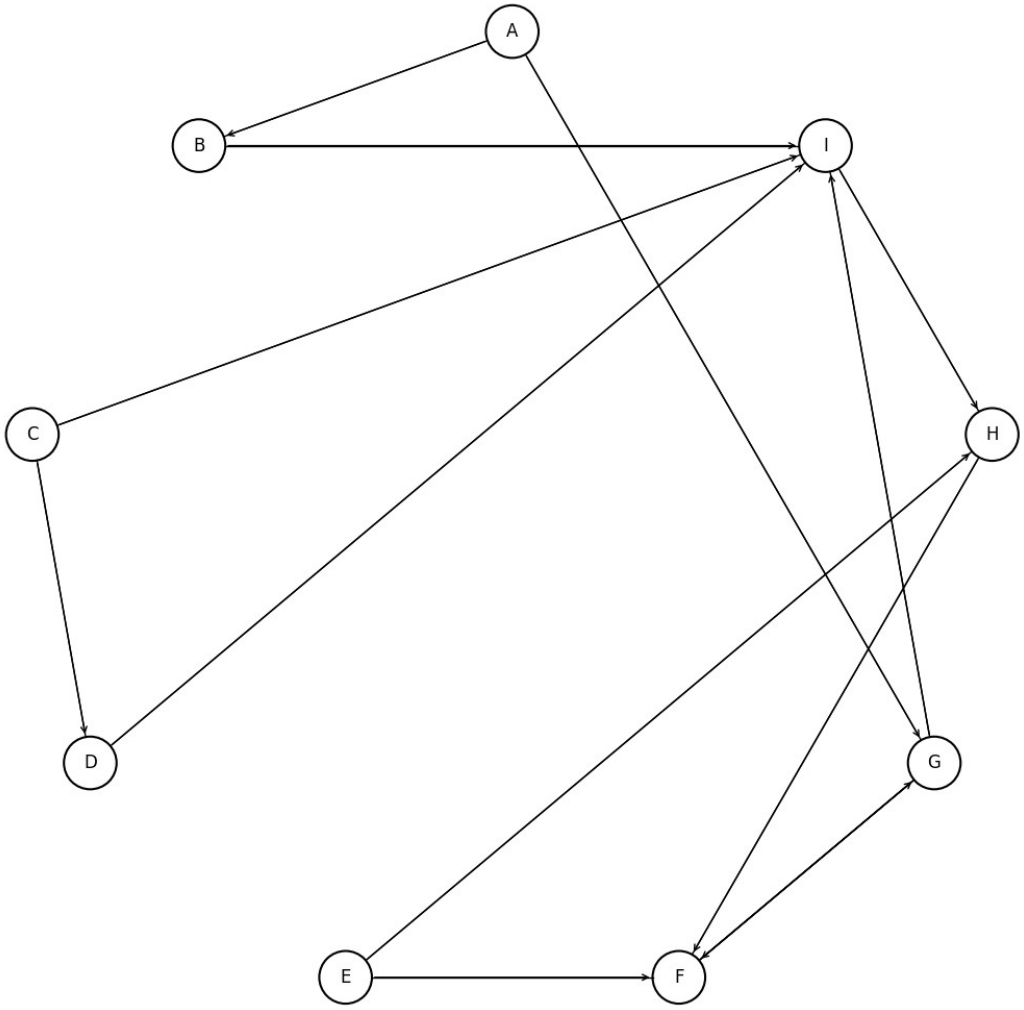
Indiquez l'ordre de traitement des sommets pour les parcours suivants :

- Parcours en largeur
- Parcours en profondeur

Le point de départ est le sommet B. Les voisins sont parcourus en ordre alphabétique.

Exercice 2

Soit le graphe non-valué et orienté suivant :



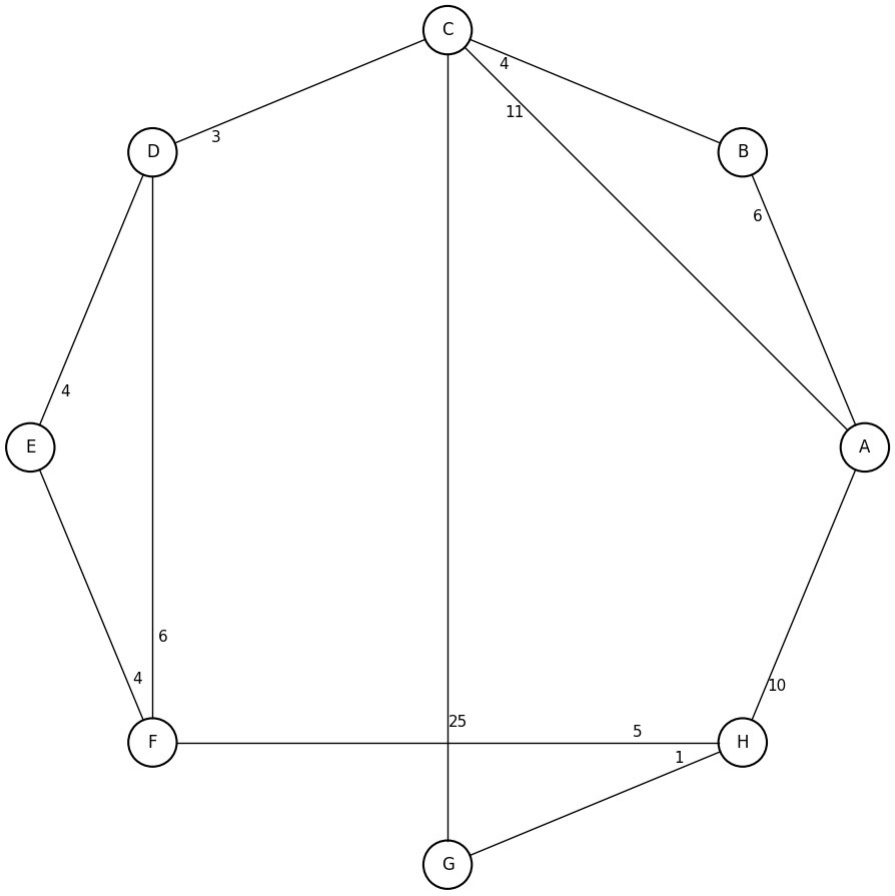
Indiquez l'ordre de traitement des sommets pour les parcours suivants :

- Parcours en largeur
- Parcours en profondeur

Faites deux premiers parcours avec le sommet E et ensuite avec le sommet A. Les voisins sont parcourus en ordre alphabétique.

Exercice 3

Soit le graphe valué et non-orienté suivant :



Appliquez l'algorithme de Dijkstra. Vous devez le faire à partir du sommet C. À la fin, indiquez clairement, pour chaque sommet, la distance finale ainsi que les prédécesseurs.

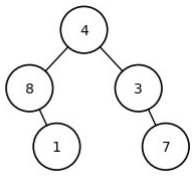
Indiquez également le chemin le moins cher pour atteindre le sommet G.

Exercice 4

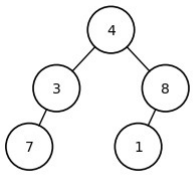
Soit les classes NoeudBinaire et ArbreBinaire suivantes :

```
class NoeudBinaire:

    def __init__(self, element, gauche=None, droite=None):
        self._element = element
        self._gauche = gauche
        self._droite = droite

    def get_element(self):
        return self._element

    def get_gauche(self):
        return self._gauche

    def set_gauche(self, noeud) -> None:
        self._gauche = noeud

    def get_droite(self):
        return self._droite

    def set_droite(self, noeud) -> None:
        self._droite = noeud

class ArbreBinaire:
    def __init__(self, racine:NoeudBinaire=None):
        self._racine = racine

    def get_racine(self) -> NoeudBinaire:
        return self._racine

    def set_racine(self, noeud:NoeudBinaire) -> None:
        self._racine = noeud
```

Ajouter la méthode inverser(self) à la classe ArbreBinaire. Cette méthode inverse tous les nœuds de l'arbre. Par exemple, si l'arbre a l'aspect suivant avant l'appel de la méthode :



Alors, il aura l'aspect suivant après l'appel de la méthode :



Vous pouvez ajouter des méthodes à la classe ArbreBinaire si vous le jugez utile.

Exercice 5

Écrivez la fonction :

```
def plus_frequent(elements:list) ->  Any
```

Cette fonction retourne la valeur dans la liste qui est la plus fréquente. C'est une liste qui peut contenir plusieurs doublons.

Par exemple, si vous avez cette liste :

6	7	6	4	4	7	7	4	7	8	1	2	7	7	3	4

Vous devez retourner 7 car il présent 6 fois. En cas d'égalité, il faut retourner la valeur dont la première occurrence est la première dans la liste.

Votre solution doit utiliser un dictionnaire. Indiquez la complexité temporelle de votre algorithme.

Exercice 6

Soit le TDA Graphe suivant :

```
class Graphe:

    # Graphe représenté par une matrice d'adjacence. On utilise un dictionnaire pour
    # retrouver l'indice d'un sommet
    def __init__(self, dirige=False, value=False):
        self._sommets = {}
        self._matrice = []
        self._dirige = dirige
        self._value = value
        if self._value:
            self._valeur_defaut = None
        else:
            self._valeur_defaut = False

    def get_sommets(self) -> list:
        return list(self._sommets.keys())

    def est_adjacent(self, sommet_1, sommet_2) -> bool:
        indice_1 = self._sommets[sommet_1]
        indice_2 = self._sommets[sommet_2]
        return self._matrice[indice_1][indice_2] not in [None, False]

    def get_poids(self, sommet_1, sommet_2):
        indice_1 = self._sommets[sommet_1]
        indice_2 = self._sommets[sommet_2]
        if not self._value:
            raise ValueError("Pas de poids dans un graphe non valué")
        return self._matrice[indice_1][indice_2]

    def ajouter_sommet(self, sommet):
        if sommet in self._sommets.keys():
            raise ValueError("Le sommet existe déjà")
        else:
            nouvel_indice = len(self._sommets)
            nouveau_tableau = [self._valeur_defaut] * (nouvel_indice + 1)
            self._sommets[sommet] = nouvel_indice
            for ligne in self._matrice:
                ligne.append(self._valeur_defaut)
            self._matrice.append(nouveau_tableau)
```

```
def ajouter_arete(self, sommet_1, sommet_2, poids=None):
    if poids == None and self._value:
        raise ValueError("Il faut un poids dans un graphe valué")
    indice_1 = self._sommets[sommet_1]
    indice_2 = self._sommets[sommet_2]

    if self._value:
        valeur = poids
    else:
        valeur = True

    self._matrice[indice_1][indice_2] = valeur
    if not self._dirige:
        self._matrice[indice_2][indice_1] =  valeur

def get_voisins(self, sommet):
    voisins = []
    for candidat in self._sommets.keys():
        if self.est_adjacent(sommet, candidat):
            voisins.append(candidat)
    return voisins
```

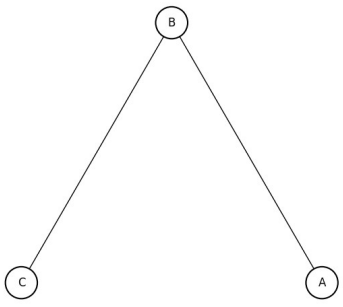
Faites la fonction suivante :

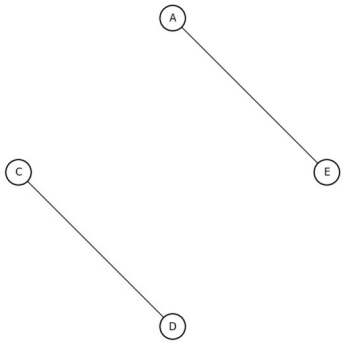
`def fusionner(graphe_1:Graphe, graphe_2:Graphe) -> Graphe:`

Cette fonction fusionne deux graphes en un seul graphe et retourne le graphe résultant. Il est possible que les deux graphes partagent des sommets semblables. Ce n'est pas une erreur, votre code doit quand même fusionner correctement les deux graphes.

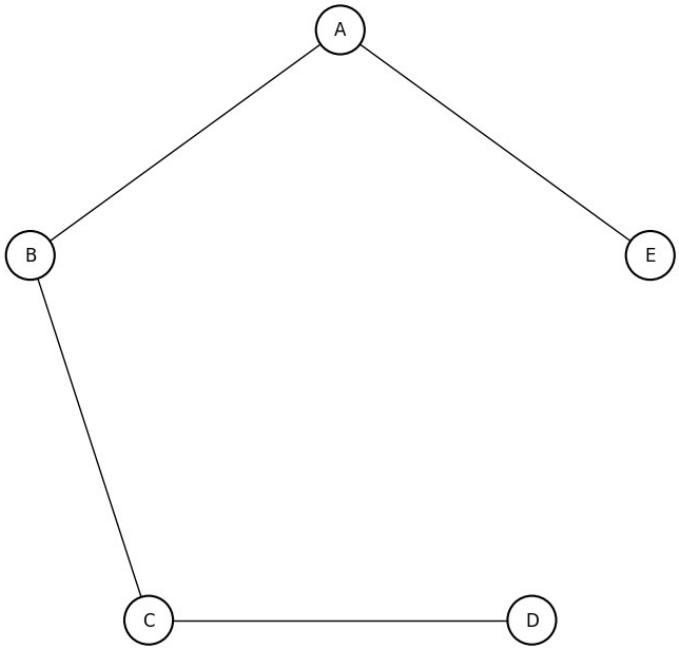
Prenez pour acquis que les deux graphes sont non-orienté et non-valué.

Par exemple, si vous avez ces graphes :





Alors, le résultat de la fusion est le suivant :



Pour cette question, vous devez utiliser seulement le TDA Graphe.