Exercice 1 : Utilisation de Graphviz

Graphviz est un logiciel qui permet de tracer graphiquement un graphe.

1- Ecrire et exécuter le script ci-dessous. Il utilise la classe *Graph* qui est importée. Dessiner ci-dessous le graphe obtenu :

```
from graphviz import Digraph , Graph
graphe = Graph()

# ----------------------------------------
graphe.node('Lyon',color='red')
graphe.edge('Lyon','Marseille','316 km')
graphe.edge('Marseille','Toulouse','405 km')
graphe.edge('Toulouse','Lyon','467 km')
graphe.view()
```

Graphe obtenu :

2- Qu'obtient-on si on utilise la classe importée *Digraph* à la place de *Graph* ? :

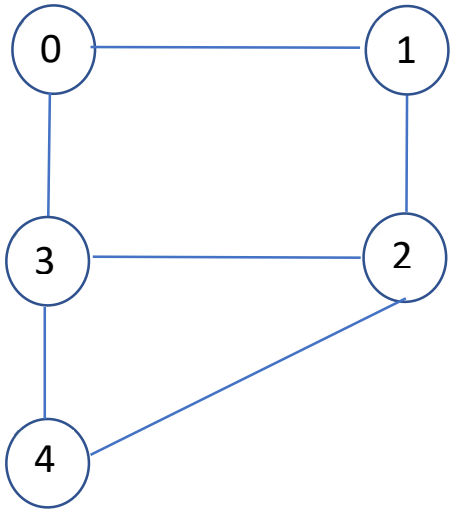
```
from graphviz import Digraph , Graph
graphe = Graph()

class G :
    def __init__(self,n) :
        self.n = n
        self.matrice = [[0 for j in range(n)] for i in range(n)]

    def ajout_arete(self,s1,s2) :




    def trace(self) :




# Main
g1 = G(5)
g1.ajout_arete(0,3)
g1.ajout_arete(0,1)
g1.ajout_arete(3,2)
g1.ajout_arete(1,2)
g1.ajout_arete(4,3)
g1.ajout_arete(4,2)

g1.trace()
```

1- Ecrire le script de la méthode *ajout_arete()*

2- Ecrire le script de la méthode *trace()*

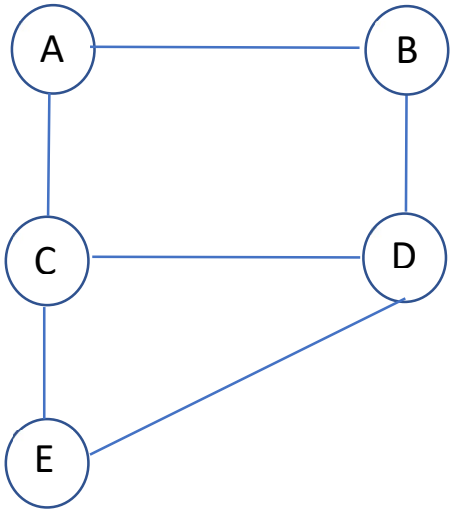
Exercice 2 : Implémentation d'un graphe en utilisant une matrice d'adjacence

L'objectif de cet exercice est d'implémenter l'arbre ci-contre en utilisant la technique des matrices d'adjacence.



Exercice 3 : Modification du code précédent pour pouvoir accepter des sommets qui soient des chaines de caractères.

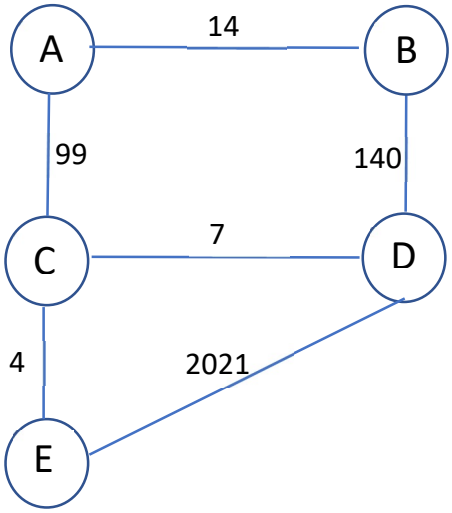
```
class G :
    def __init__(self,n) :
        self.n = n
        self.indices = []
        self.matrice = [[0 for j in range(n)] for i in range(n)]

    def ajout_arete(self,s1,s2) :
        [                                                    ]

    def trace(self) :
        graphe.clear()
        for i in range(self.n)  :
            for j in range(i , self.n) :
                if self.matrice[i][j] == 1 :
                    graphe.edge(str(self.indices[i]),str(self.indices[j]))
        graphe.view()


# Main
g1 = G(5)
g1.ajout_arete('A','C')
g1.ajout_arete('A','B')
g1.ajout_arete('C','D')
g1.ajout_arete('B','D')
g1.ajout_arete('E','D')
g1.ajout_arete('E','C')
g1.trace()
```



⇨ Ecrire le script de la méthode *ajout_arete()*

Exercice 4 : Modification du code précédent pour pouvoir accepter le traitement d'un graphe pondéré

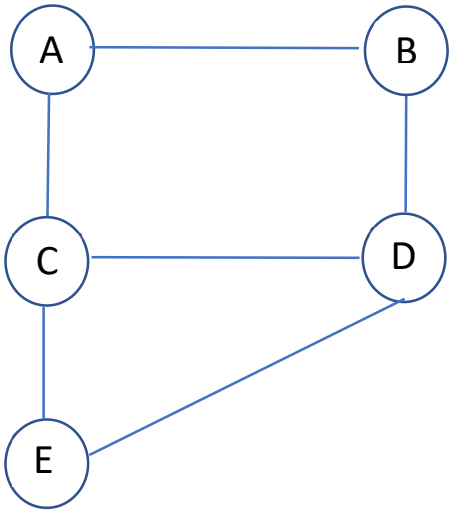
```
class G :
    def __init__(self,n) :
        self.n = n
        self.indices = []
        self.matrice = [[0 for j in range(n)] for i in range(n)]

    def ajout_arete(self,s1,s2,d) :
        [                                                    ]

    def trace(self) :
        graphe.clear()
        for i in range(self.n) :
            for j in range(self.n) :
                if self.matrice[i][j] != 0 :
                    graphe.edge(str(self.indices[i]),str(self.indices[j]),str(self.matrice[i]
[j]))
        graphe.view()

# Main
g1 = G(5)
g1.ajout_arete('A','C',99)
g1.ajout_arete('A','B',14)
g1.ajout_arete('C','D',7)
g1.ajout_arete('B','D',140)
g1.ajout_arete('E','D',2021)
g1.ajout_arete('E','C',4)
g1.trace()
```



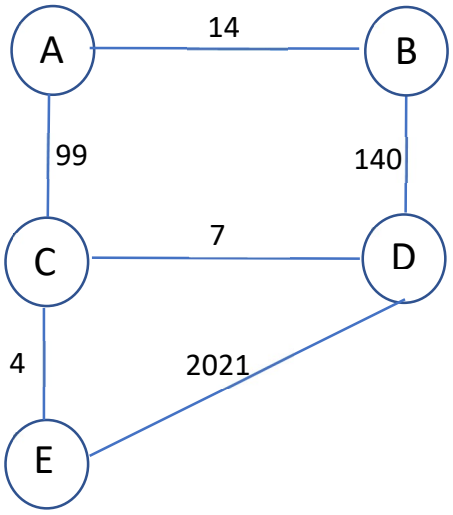
Exercice 5 : Implémentation d'un graphe en utilisant un dictionnaire d'adjacence

```
class G :
    def __init__(self) :
        self.dic = {}

    def ajout_arete(self,s1,s2) :
        [redacted]

    def trace(self) :
        graphe.clear()
        deja_trace = []
        [redacted]
        graphe.view()

# Main
g1 = G()
g1.ajout_arete('A','C')
g1.ajout_arete('A','B')
g1.ajout_arete('C','D')
g1.ajout_arete('B','D')
g1.ajout_arete('E','D')
g1.ajout_arete('E','C')
g1.trace()
```

```
dic = { "A" : [ "B" ,"C" ],
        "B" : [ "A", "D"],
        "C" : ["A", "D","E"],
        "D" : ["B", "C","E"],
        "E" : ["C", "D"]
}
```



Exercice 6 : Modification du code précédent pour pouvoir implémenter un graphe pondéré.

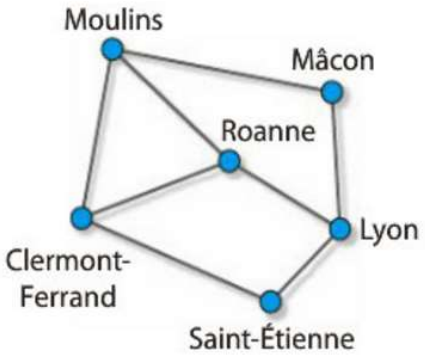



```
dic = { "A" : { "B": 14 ,"C": 99 },
        "B" : { "A": 14 , "D": 140 },
        "C" : { "A": 99 , "D":7, "E": 4 },
       "D" : { "E": 2021 ,"B": 140,"C": 7 },
        "E" : { "C" :4, "D" :2021 }
```

```
# Main
g1 = G()
g1.ajout_arete('A','C',99)
g1.ajout_arete('A','B',14)
g1.ajout_arete('C','D',7)
g1.ajout_arete('B','D',140)
g1.ajout_arete('E','D',2021)
g1.ajout_arete('E','C',4)
g1.trace()
```

Exercice 7 : Itinéraire routier

On donne ci-dessous le graphe représentant un réseau routier :



Les itinéraires calculés avec un GPS donnent :

- Moulins – Macon : 141 km en 1h47
- Moulins – Roanne : 98 km en 1h34
- Moulins – Clermont-Ferrand : 103 km en 1h23
- Lyon – Mâcon : 73 km en 0h52
- Lyon - Roanne : 86 km en 1h26
- Lyon – Saint-Etienne : 63 km en 0h48
- Clermont-Ferrand - Saint-Etienne : 147 km en 1h28
- Clermont-Ferrand - Roanne : 125 km en 1h19

1- Représenter deux graphes pondérés traduisant cette situation, l'un ayant ses arêtes étiquetées avec les distances entre villes, l'autre ayant ses arêtes étiquetées avec les temps de parcours en minutes.

2- Marion habite Lyon et souhaite se rendre dans sa maison de campagne à Moulins le plus vite possible. Quel itinéraire va-t-elle choisir ?

3- Pour le retour, elle décide de revenir sur Lyon en économisant au maximum son carburant et donc en empruntant une route qui lui permette de parcourir le minimum de kilomètres. Quel itinéraire va-t-elle choisir ?

4- Implémenter ces 2 graphes en utilisant le dernier code mise au point.