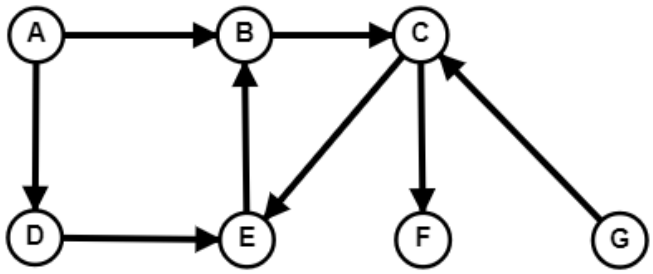
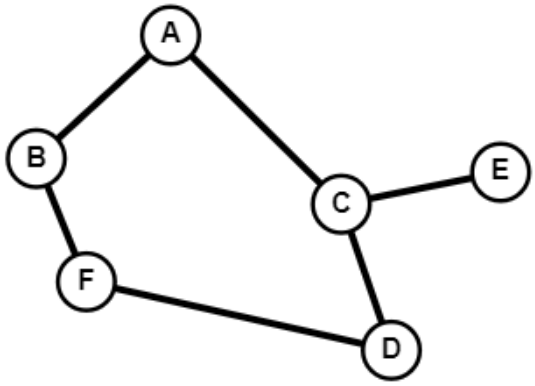
Exercices - Graphes et algorithmes

Exercice 1

Voici deux graphes que l'on appelera respectivement `G1` et `G2`.



graphe `G1`



graphe `G2`

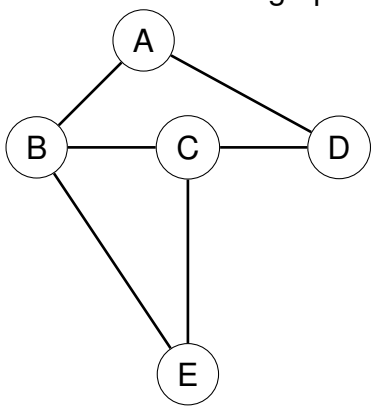
1. Lequel est non orienté ?
2. Pour le graphe non orienté :
   - Donner deux sommets adjacents et deux sommets non adjacents.
   - Donner les voisins de A ?
   - Quels sont les degrés des sommets B, C et E ?
   - S'il y en a, donner un cycle de ce graphe.
   - Donner toutes les chaînes entre les sommets A et D.
3. Pour le graphe orienté :
   - Donner les successeurs et les prédecesseurs des sommets A et C.
   - S'il y en a, donner un chemin entre G et B. Et entre B et D ?
   - S'il y en a, donner un circuit de ce graphe.
   - Quel est le sommet dont le degré est le plus grand ?
4. Donner la matrice d'adjacence de chacun de ces graphes (on prendra les indices des sommets dans l'ordre alphabétique).
5. Donner la représentation de chacun de deux graphes sous la forme d'un dictionnaire de liste de successeurs.
6. On considère le graphe G3 défini par la matrice d'adjacence $M3$
   - Le graphe G3 est-il orienté ?
   - dessiner le graphe G3.

$$M3 = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

7. On considère le graphe G4 défini par le dictionnaire suivant :
   - Le graphe G4 est-il orienté ?
   - Dessiner le graphe G4.

```
G4 = {
    "A": ["D", "C"],
    "B": ["C", "D", "E"],
    "C": ["A", "B", "F"],
    "D": ["A", "B"],
    "E": ["B"],
    "F": ["C"]
}
```

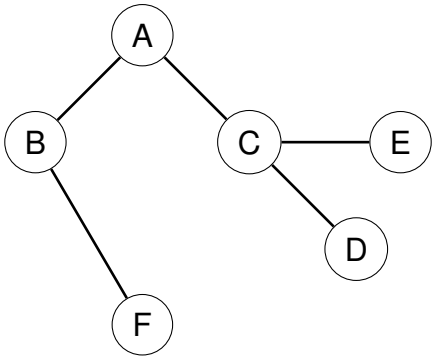
Exercice 2 Soit le graphe non-orienté suivant :



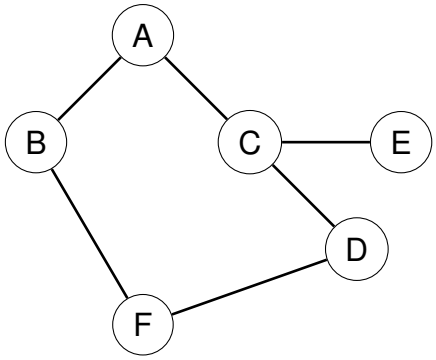
1. Appliquez l'algorithme du **parcours en largeur d'abord** au graphe ci-dessus. Le 'point de départ' de notre parcours sera le sommet A. Vous noterez les sommets atteints à chaque étape ainsi que les sommets présents dans la file `f`. Vous pourrez aussi, à chaque étape, donner les changements d'états (`True` or `False`) des sommets.
2. Appliquez l'algorithme du **parcours en profondeur** d'abord au graphe ci-dessus (d'abord avec l'algorithme récursif puis ensuite avec l'algorithme non récursif). Le 'point de départ' de notre parcours sera le sommet A. Vous noterez les sommets atteints à chaque étape ainsi que les sommets présents dans la pile `p`. Vous pourrez aussi, à chaque étape, donner les changements d'états des sommets.

Exercice 3

1. Pour chacun des deux graphes suivants `g1` et `g2`, appliquez l'algorithme de détection d'un cycle à partir du sommet A. Vous détaillerez à chaque étapes l'état de structure de données utilisée pour stocker les sommets.

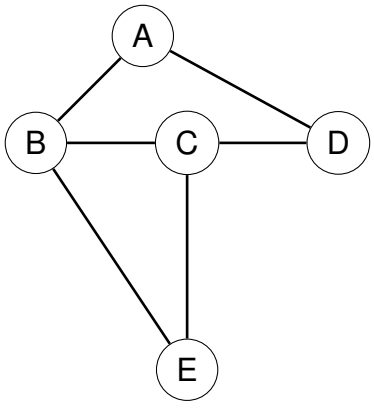


Graphe `g1`



Graphe `g2`

2. Appliquez l'algorithme permettant de trouver une chaine entre le sommet de départ A et le sommet d'arrivée E au graphe ci-dessous . Vous détaillerez à chaque étapes l'état de structure de données utilisée pour stocker les sommets.



Exercice 4

Vous avez décidé de développer un réseau social à l'échelle du lycée. Afin d'effectuer des tests, vous décidez de limiter votre réseau à social à 6 utilisateurs que vous décidez de nommer : A, B, C, D, E et F. À un instant t, voici l'état de votre réseau social :

- A et B sont amis
- A et C sont amis
- A et D sont amis
- B et E sont amis
- B et F sont amis
- E et F sont amis

1. Vous décidez de représenter l'état de votre réseau social à l'instant t par un graphe non orienté G. Les personnes (A, B, C,. . .) seront les sommets du graphe G. Une relation « x et y sont amis » sera une arête de G. Représentez graphiquement le graphe G.
2. Représentez la matrice d'adjacence du graphe G (A est associé à l'indice 1 de la matrice, B à l'indice 2, C à l'indice 3, etc.)
3. Le parcours [A, B, C, D, E, F] est-il un parcours « en profondeur d'abord » ou un parcours « en largeur d'abord » ? Justifiez votre réponse
4. On donne ci-dessous l'algorithme (en pseudo-code) permettant d'obtenir le parcours en « largeur d'abord » d'un graphe G. Complétez cet algorithme (si possible sans vous aider du cours)

```
VARIABLE
G : un graphe
s : sommet (origine)
u : sommet
v : sommet
f : file (initialement vide)

DEBUT
s.couleur <-- noir
enfiler (s,f)
tant que f non vide :
    u <-- .....
    pour chaque sommet v adjacent au sommet .... :
        si v.couleur n'est pas ..... :
            v.couleur <-- noir
            enfiler(...,f)
        fin si
    fin pour
fin tant que
FIN
```

Exercice 5

Soit le graphe `g1` représenté comme suit :

```
g1 = {'A':['B','C'], 'B':['A'], 'C':['A','D'], 'D':['C']}
```

1. Représentez graphiquement le graphe `g1`.
2. Représentez le graphe `g1` sous la forme d'une matrice d'adjacence.
3. Représentez cette matrice d'adjance en python.

4. Ce graphe est-il complet ? connexe ? orienté ?
5. Le graphe `g1` contient-il des cycles ? si oui donnez un exemple.
6. Soit le programme Python suivant :

```
def myst(G,s):
    noir = []
    pile = [s]
    while len(pile) > 0 :
        u = pile.pop()
        if u not in noir :
            noir.append(u)
            for v in G[u]:
                pile.append(v)
    return noir


L = myst(g1,'A')
```

   (a) Quel est l'utilité de la liste `noir` ?
   (b) Que vaut `L` après l'exécution de ce programme ? En déduire ce que fait la fonction `myst` ?

7. Complétez le programme Python suivant (la fonction cycle prend en paramètre un graphe G et retourne True si le graphe G possède un cycle et False dans le cas contraire), si possible sans vous aider du cours.

```
def cycle(G):
    s = random.choice(list(G.keys()))
    p = []
    p.append(s)
    noir=[]
    while len(p)>0:
        u = p.pop()
        for v in ....:
            if v not in noir:
                p.append(....)
        if u in ....:
            return True
        else :
            noir.append(u)
    return ....
```

Exercice 6 Implémentation par matrice d'adjacence (Graphe non orienté)

Type abstrait `GrapheNonOriente`

On peut doter le type abstrait des constructeurs suivants :

- `faire_graphe(sommets)` pour construire un graphe (sans arêtes) à partir de la liste `sommets` de ses sommets.
- `ajouter_arete(G, x, y)` pour ajouter une arête entre les sommets `x` et `y` du graphe `G`.

Pour pouvoir parcourir un graphe non orienté, on a besoin d'accéder à la liste des sommets et d'accéder aux sommets voisins d'un sommet donné.

- `sommets(G)` pour accéder à la liste des sommets du graphe `G`.
- `voisins(G, x)` pour accéder à la liste des voisins du sommet `x` du graphe `G`.

Représentation par matrice d'adjacence

On choisit de créer une classe `GrapheNoMa` pour implémenter un graphe non orienté par sa matrice d'adjacence à partir de la liste `sommets` de ses sommets (au sens list de Python). En voici une implémentation incomplète puisqu'il manque la méthode `voisins(self, x)`.

```
class GrapheNoMa:
    def __init__ (self, sommets):
        self.som = sommets
        self.dimension = len(sommets)
        self.adjacence = [[0 for i in range(self.dimension)] for j in range(self.
            dimension)]

    def ajouter_arete(self, x, y):
        i = self.som.index(x)
        j = self.som.index(y)
        self.adjacence[i][j] = 1
        self.adjacence[j][i] = 1

    def sommets(self):
        return self.som

    def voisins(self, x):
        pass
```

1. Combien d'attributs possèdent les (objets) graphes de cette classe ? Quels sont leurs noms ? Lors de la création d'un objet `GrapheNonOriente` de cette classe, que contient la matrice d'adjacence ? Est-ce normal ?
2. Quelle instruction permet de créer un objet `g1` de cette classe contenant les sommets "a", "b", "c", et "d" ?
3. Quelles instructions permettent d'accéder aux attributs de du graphe `g1` ?
4. Expliquer le rôle de chaque ligne de la méthode `ajouter_arete`. N'hésitez pas à consulter la documentation de Python sur les listes.
5. Ecrivez les instructions pour ajouter les arêtes ("a","b"), ("a","c") et ("c","d"). Vous écrirez ensuite que la matrice d'adjacence correspondante.
6. Complétez la méthode voisins à la classe.
7. Quelle instruction écrire pour afficher la liste des voisins du sommet "a" ? Et pour les autres sommets ?

Exercice 7 Extrait BAC 2024 sujet 1 ex 3

La société CarteMap développe une application de cartographie-GPS qui permettra aux automobilistes de définir un itinéraire et d'être guidés sur cet itinéraire. Dans le cadre du développement d'un prototype, la société CarteMap décide d'utiliser une carte fictive simplifiée comportant uniquement 7 villes : A, B, C, D, E, F et G et 9 routes (toutes les routes sont considérées à double sens). Voici une description de cette carte :

- A est relié à B par une route de 4 km de long ;
- A est relié à E par une route de 4 km de long ;
- B est relié à F par une route de 7 km de long ;
- B est relié à G par une route de 5 km de long ;
- C est relié à E par une route de 8 km de long ;
- C est relié à D par une route de 4 km de long ;

- D est relié à E par une route de 6 km de long ;
- D est relié à F par une route de 8 km de long ;
- F est relié à G par une route de 3 km de long.

1. Représenter ces villes et ces routes sur sa copie en utilisant un graphe pondéré, nommé G1.
2. Déterminer le chemin le plus court possible entre les villes A et D.
3. Définir la matrice d'adjacence du graphe G1 (en prenant les sommets dans l'ordre alphabétique).

Dans la suite de l'exercice, on ne tiendra plus compte de la distance entre les différentes villes et le graphe, non pondéré et représenté ci-contre, sera utilisé.
Chaque sommet est une ville, chaque arête est une route qui relie deux villes.

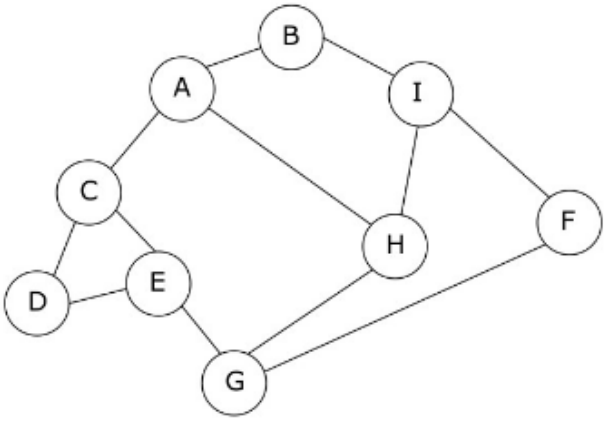


Figure 1. Graphe G2

4. Proposer une implémentation en Python du graphe G2 à l'aide d'un dictionnaire.
5. Proposer un parcours en largeur du graphe G2 en partant de A.

La société CarteMap décide d'implémenter la recherche des itinéraires permettant de traverser le moins de villes possible. Par exemple, dans le cas du graphe G2, pour aller de A à E, l'itinéraire A-C-E permet de traverser une seule ville (la ville C), alors que l'itinéraire A-H-G-E oblige l'automobiliste à traverser 2 villes (H et G). Le programme Python suivant a donc été développé (programme p1) :

```
1 tab_itineraires=[]
2 def cherche_itineraires(G, start, end, chaine=[]):
3   chaine = chaine + [start]
4   if start == end:
5     return chaine
6   for u in G[start]:
7     if u not in chaine:
8       nchemin = cherche_itineraires(G, u, end, chaine)
9       if len(nchemin) != 0:
10         tab_itineraires.append(nchemin)
11  return []
12
13 def itineraires_court(G,dep,arr):
14   cherche_itineraires(G, dep, arr)
15   tab_court = ...
16   mini = float('inf')
17   for v in tab_itineraires:
18     if len(v) <= ... :
19       mini = ...
20   for v in tab_itineraires:
21     if len(v) == mini:
22       tab_court.append(...)
23   return tab_court
```

La fonction `itineraires_court` prend en paramètre un graphe `G`, un sommet de départ `dep` et un sommet d'arrivée `arr`. Cette fonction renvoie une liste Python contenant tous les itinéraires pour aller de `dep` à `arr` en passant par le moins de villes possible.
Exemple (avec le graphe G2) :

```
itineraires_court(G2, 'A', 'F')
>>> [['A', 'B', 'I', 'F'], ['A', 'H', 'G', 'F'], ['A', 'H', 'I',
'F']]
```

On rappelle les points suivants :

- la méthode `append` ajoute un élément à une liste Python ; par exemple, `tab.append(el)` permet d'ajouter l'élément `el` à la liste Python `tab` ;
- en python, l'expression `['a'] + ['b']` vaut `['a', 'b']` ;
- en python `float('inf')` correspond à l'infini.

6. Expliquer pourquoi la fonction `cherche_itineraires` peut être qualifiée de fonction récursive.
7. Expliquer le rôle de la fonction `cherche_itineraires` dans le programme p1.
8. Compléter la fonction `itineraires_court`.

Les ingénieurs sont confrontés à un problème lors du test du programme p1. Voici les résultats obtenus en testant dans la console la fonction `itineraires_court` deux fois de suite (sans exécuter le programme entre les deux appels à la fonction `itineraires_court`) :

```
execution du programme p1

itineraires_court(G2, 'A', 'E')
>>> [['A', 'C', 'E']]

itineraires_court(G2, 'A', 'F')
>>> [['A', 'C', 'E']]
```

alors que dans le cas où le programme p1 est de nouveau exécuté entre les 2 appels à la fonction `itineraires_court`, on obtient des résultats corrects :

```
execution du programme p1

itineraires_court(G2, 'A', 'E')
>>> [['A', 'C', 'E']]

execution du programme p1

itineraires_court(G2, 'A', 'F')
>>> [['A', 'B', 'I', 'F'], ['A', 'H', 'G', 'F'], ['A', 'H', 'I',
'F']]
```

9. Donner une explication au problème décrit ci-dessus. Vous pourrez vous appuyer sur les tests donnés précédemment.

Exercice 8 Extrait BAC 2024 sujet 3 ex 2

Dans cet exercice, on modélise un groupe de personnes à l'aide d'un graphe. Le groupe est constitué de huit personnes (Anas, Emma, Gabriel, Jade, Lou, Milo, Nina et Yanis) qui possèdent entre elles les relations suivantes :

- Gabriel est ami avec Jade, Yanis, Nina et Milo ;
- Jade est amie avec Gabriel, Yanis, Emma et Lou ;
- Yanis est ami avec Gabriel, Jade, Emma, Nina, Milo et Anas ;
- Emma est amie avec Jade, Yanis et Nina ;
- Nina est amie avec Gabriel, Yanis et Emma ;
- Milo est ami avec Gabriel, Yanis et Anas ;
- Anas est ami avec Yanis et Milo ;
- Lou est amie avec Jade.

Partie A : Matrice d'adjacence

On choisit de représenter cette situation par un graphe dont les sommets sont les personnes et les arêtes représentent les liens d'amitié.

1. Dessiner sur votre copie ce graphe en représentant chaque personne par la première lettre de son prénom entourée d'un cercle et où un lien d'amitié est représenté par un trait entre deux personnes.

   Une matrice d'adjacence est un tableau à deux entrées dans lequel on trouve en lignes et en colonnes les sommets du graphe.
   Un lien d'amitié sera représenté par la valeur 1 à l'intersection de la ligne et de la colonne qui représentent les deux amis alors que l'absence de lien d'amitié sera représentée par un 0.

2. Recopier et compléter l'implémentation de la déclaration de la matrice d'adjacence du graphe.

```
# sommets : G, J, Y, E, N, M, A, L
matrice_adj = [[0, 1, 1, 0, 1, 1, 0, 0], # G
[....................], # J
[....................], # Y
[....................], # E
[....................], # N
[....................], # M
[....................], # A
[....................]] # L
```

   On dispose de la liste suivante qui identifie les sommets du graphe :

```
sommets = ['G', 'J', 'Y', 'E', 'N', 'M', 'A', 'L']
```

   On dispose d'une fonction `position(l, s)` qui prend en paramètres une liste de sommets `l` et un nom de sommet `s` et qui renvoie la position du sommet `s` dans la liste `l` s'il est présent et `None` sinon.

3. Indiquer quel seront les retours de l'exécution des instructions suivantes :

```
>>> position(sommets, 'G')
>>> position(sommets, 'Z')
```

4. Recopier et compléter le code de la fonction `nb_amis(L, m, s)` qui prend en paramètres une liste de noms de sommets `L`, une matrice d'adjacence `m` d'un graphe et un nom de sommet `s` et qui renvoie le nombre d'amis du sommet `s` s'il est présent dans `L` et `None` sinon.

```
1 def nb_amis(L, m, s):
2   pos_s = ...
3   if pos_s == None:
4     return ...
5   amis = 0
6   for i in range(len(m)):
7     amis += ...
8   return ...
```

5. Indiquer quel est le retour de l'exécution de la commande suivante :

```
>>> nb_amis(sommets, matrice_adj, 'G')
```

Partie B : Dictionnaire de listes d'adjacence

6. Dans un dictionnaire Python `c : v`, indiquer ce que représentent `c` et `v`.

On appelle graphe le dictionnaire de listes d'adjacence associé au graphe des amis. On rappelle que Gabriel est ami avec Jade, Yanis, Nina et Milo.

```
graphe = {'G' : ['J', 'Y', 'N', 'M'],
          'J' : ...
          ...
         }
```

7. Recopier et compléter le dictionnaire de listes d'adjacence graphe sur votre copie pour qu'il modélise complètement le groupe d'amis.
8. Écrire le code de la fonction `nb_amis(d, s)` qui prend en paramètres un dictionnaire d'adjacence `d` et un nom de sommet `s` et qui renvoie le nombre d'amis du nom de sommet `s`. On suppose que `s` est bien dans `d`.
Par exemple :

```
>>> nb_amis(graphe, 'L')
1
```

Milo s'est fâché avec Gabriel et Yanis tandis qu'Anas s'est fâché avec Yanis. Le dictionnaire d'adjacence du graphe qui modélise cette nouvelle situation est donné ci-dessous :

```
graphe = {'G' : ['J', 'N'],
          'J' : ['G', 'Y', 'E', 'L'],
          'Y' : ['J', 'E', 'N'],
          'E' : ['J', 'Y', 'N'],
          'N' : ['G', 'Y', 'E'],
          'M' : ['A'],
          'A' : ['M'],
          'L' : ['J']
}
```

Pour établir la liste du cercle d'amis d'un sommet, on utilise un parcours en profondeur du graphe à partir de ce sommet. On appelle cercle d'amis de *Nom* toute personne atteignable dans le graphe à partir de *Nom*.

9. Donner la liste du cercle d'amis de Lou.

Un algorithme possible de parcours en profondeur de graphe est donné ci-dessous :

```
visites = liste vide des sommets deja visites

fonction parcours_en_profondeur(d, s)
  ajouter s a la liste visites
  pour tous les sommets voisins v de s :
    si v n'est pas dans la liste visites :
      parcours_en_profondeur(d, v)
  retourner la liste visites
```

10. Recopier et compléter le code de la fonction `parcours_en_profondeur(d, s)` qui prend en paramètres un dictionnaire d'adjacence `d` et un sommet `s` et qui renvoie la liste des sommets issue du parcours en profondeur du graphe modélisé par `d` à partir du sommet `s`.

```
1 def parcours_en_profondeur(d, s, visites = []):
2   ...
3   for v in d[s]:
4     ...
5       parcours_en_profondeur(d, v)
6   ...
```