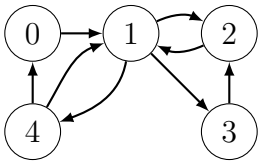
Graphes

Exercice 1

1. Dessiner tous les graphes non orientés (simples, c'est à dire sans boucle, c'est à dire sans arête reliant un sommet à lui-même) ayant exactement 3 sommets.
2. Dessiner au moins neuf graphes orientés ayant exactement 3 sommets.

Exercice 2
On considère le graphe suivant :

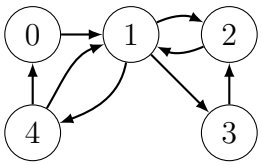


1. Donner le degré de chacun des sommets.
2. Donner 3 cycles différents de ce graphe.

Exercice 3
Tracer les graphes associés aux matrices d'adjacence données, les sommets étant numérotés à partir de 0.

$$M_1 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \end{array} \qquad M_2 = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \end{pmatrix} \qquad M_3 = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

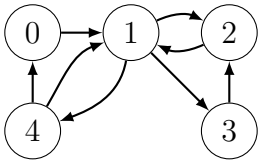
Exercice 4
Écrire la matrice d'adjacence du graphe ci-dessous :



Exercice 5
Donner le graphe associé à liste d'adjacence suivante :

```
{0 : [5,6], 1 : [], 2 : [0,6], 3 : [3,6], 4 : [], 5 : [1,2], 6 : [3]}
```

Exercice 6
Écrire la liste d'adjacence du graphe ci-dessous :



⚠ Comme pour les fiches précédentes tester dans chacun les exercices suivants les méthodes et fonctions sur les classes en créant des objets et en utilisant `assert`.

Exercice 7 (classe Graphe pour les graphes orientés définis par matrice d'adjacence)
Définir, pour les graphes orientés, dont on considérera les sommets nommés par les indices de `0` à `n-1` (où `n` est l'ordre du graphe), une classe `Graphe` contenant :

- La fonction d'initialisation `__init__(self,n)` qui crée le graphe avec un attribut `adj` qui est la matrice d'adjacence (nulle au départ) et un attribut `n`, ordre du graphe donné en argument ;
- La méthode `ajouter_arc(self,s1,s2)` qui permet d'ajouter un arc entre le sommet `s1` et le sommet `s2` ;
- La méthode `arc(self,s1,s2)` qui retourne `True` s'il y a un arc depuis `s1` vers `s2`, `False` sinon ;
- La méthode `ordre(self)` qui retourne l'ordre du graphe ;
- La méthode `sommets(self)` qui retourne la liste des sommets du graphe
- La méthode `voisins(self,s)` qui retourne la liste des sommets adjacents au sommet `s`
- La méthode `degre(self,s)` qui retourne le degré du sommet `s`.
- La méthode `liste_adjacence(self)` qui retourne la liste d'adjacence du graphe (sous forme de dictionnaire) ;
- La méthode `matrice_adjacence(self)` qui retourne la matrice d'adjacence du graphe (sous forme de liste de listes) ;

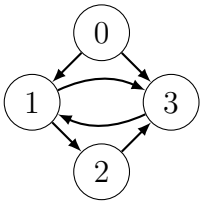
Effectuer quelques tests pour vérifier le fonctionnement de ces diverses méthodes.
Facultatif : Refaire l'exercice en permettant d'avoir des noms quelconques pour les sommets (de type quelconque, que ce soit `str` ou `int` par exemple). On pourra gérer cela à l'aide d'un dictionnaire et utiliser celui-ci pour traduire entre nom et indice, la liste (supposée ordonnée) des noms de sommets (supposés tous uniques) étant donnée en paramètre lors de l'initialisation au lieu de `n`.

Exercice 8 (classe Graphe pour les graphes non orientés définis par liste d'adjacence)
Définir (puis tester là encore), pour les graphes non orientés, dont on considérera les sommets nommés de manière quelconque, une classe `Graphe` contenant :

- La fonction d'initialisation `__init__(self)` qui crée le graphe avec seulement un attribut `adj` qui est la liste d'adjacence, de type dictionnaire (vide au départ) ;
- La méthode `ajouter_sommet(self,s)` qui permet d'ajouter le sommet `s`, si celui-ci n'existe pas déjà, au graphe ;
- La méthode `ajouter_arete(self,s1,s2)` qui permet d'ajouter une arête entre le sommet `s1` et le sommet `s2` ;
- La méthode `arete(self,s1,s2)` qui retourne `True` s'il y a une arête entre `s1` et `s2`, `False` sinon ;
- Les méthodes `ordre(self)`, `sommets(self)`, `voisins(self,s)`, `degre(self,s)`, `liste_adjacence(self)` et `matrice_adjacence(self)` définies dans l'exercice précédent.

Exercice 9 (Facultatif)
Ajouter, dans les classes `Graphe` définies précédemment, les méthodes suivantes :

1. `nb_arcs(self)` (et `nb_aretes(self)`) qui retourne le nombre d'arcs (resp. d'arêtes) du graphe ;
2. `supprimer_arc(self,s1,s2)` (et `supprimer_arete(self,s1,s2)`) qui supprime l'arc (resp. l'arête) entre `s1` et `s2` s'il existe, qui ne fait rien sinon.
3. `__str__(self)` qui retourne une chaîne de caractère décrivant le graphe. Par exemple, si le graphe `G` est celui donné ci-contre, l'instruction `print(G)` doit alors afficher :




```
0 [1, 3]
1 [2, 3]
2 [3]
3 [1]
```

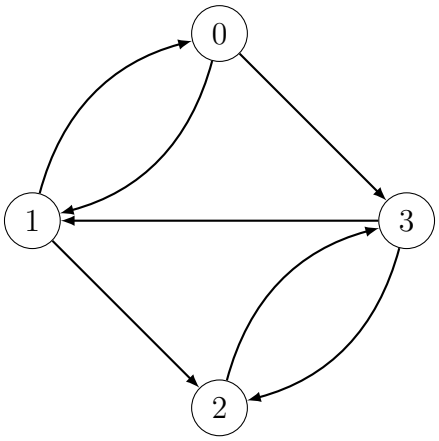
Ou bien :

```
0 -> 1 3
1 -> 2 3
2 -> 3
3 -> 1
```

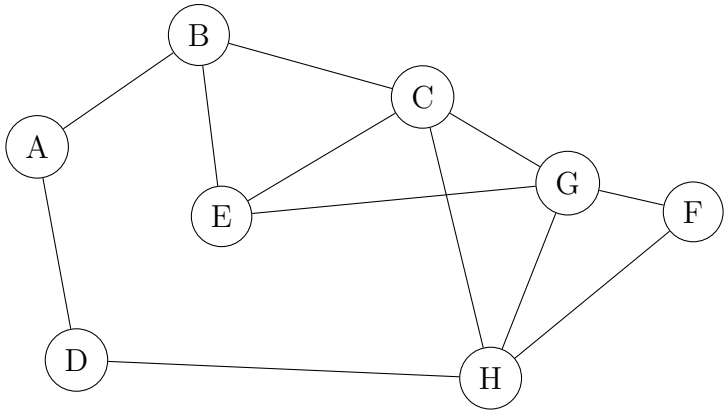
Exercice 10

Lors des parcours, les voisins seront toujours traités selon l'ordre (lexicographique) croissant.

1. Déterminer la liste ordonnée de la liste des sommets vus lors du parcours en profondeur dans le graphe suivant, en partant de chacun des sommets



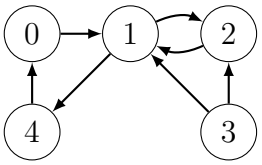
2. Même chose pour le graphe suivant, en partant du sommet A, de C puis de F.



Exercice 11

On considère la fonction `mystere` ci-dessous et le graphe `g` ci-contre.

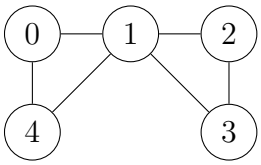
```
def mystere(g, u, v):
    vus = parcours_profondeur(g, u)
    return v in vus
```



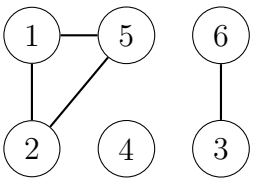
1. Quelle est le résultat de `mystere(g, 0, 4)` ?
2. Même question pour `mystere(g, 0, 3)`.
3. Décrire en une phrase le résultat de `mystere(g, u, v)` pour `u` et `v` deux sommets de `g`.

Exercice 12

On peut se servir d'un parcours en profondeur pour déterminer si un graphe *non orienté* est *connexe*, c'est à dire si tous ses sommets sont reliés entre eux par des chemins.



Graphe connexe



Graphe non connexe

Pour cela, il suffit de faire un parcours en profondeur et de vérifier que tous les sommets ont bien été visités par ce parcours.

En utilisant la méthode `sommets` de la classe `Graphe` et la fonction `parcours_profondeur`, écrire une fonction `est_connexe` qui réalise cet algorithme.

Exercice 13

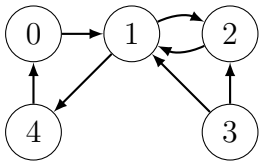
Dans cet exercice, on se propose d'utiliser un parcours en profondeur pour *construire* un chemin entre deux sommets lorsque c'est possible.

On le fait avec deux fonctions :

1. La fonction `parcours_chemin` est très semblable à la fonction `parcours_profondeur`. Elle prend un paramètre supplémentaire `org` (pour origine) qui est le sommet qui permis d'atteindre `s` et l'argument `vus` n'est plus une liste mais un *dictionnaire* qui associe à chaque sommet visité le sommet qui a permis de l'atteindre.

   Pour le graphe ci-contre, un tel parcours d'origine le sommet `1` donne le dictionnaire `vus` suivant :

   ```
   {1 : None, 2 : 1, 4 : 1, 0 : 4}
   ```

   (a) Pour le graphe ci-dessus, déterminer le dictionnaire `vus` qui sera obtenu par la fonction `parcours_chemin` avec pour origine le sommet `2`.

   (b) Compléter la fonction récursive ci-dessous afin qu'elle réalise ce parcours.

   Remarquer que cette fonction ne retourne rien mais qu'elle modifie le dictionnaire `vus` au fur et à mesure.

```
def parcours_chemin(g, vus, org, s):
    '''parcours depuis le sommet s, en venant de org'''
    if s not in vus:
        vus[s] = ........
        for v in g.voisins(s):
            ........
```

2. La deuxième fonction `chemin(g, u, v)` permet de construire le chemin entre `u` et `v` s'il existe. Pour cela on « remonte » le dictionnaire `vus` obtenu avec `parcours_chemin`.

   Par exemple, si l'on veut trouver un chemin du sommet `1` au sommet `0` et que notre dictionnaire obtenu à partir d'un parcours d'origine le sommet `1` est

   `{1 : None, 2 : 1, 4 : 1, 0 : 4}`, « remonter » le dictionnaire donnera `0 4 1 None`, ce qui donne le chemin `1 -> 4 -> 0`.

   Compléter la fonction ci-dessous pour qu'elle réalise cet algorithme.

   Elle retourne la liste des sommets parcourus, pour l'exemple elle retourne donc `[1,4,0]`.

```
def chemin(g, u, v):
    '''un chemin de u a v, le cas échéant, None sinon'''
    vus = {}
    parcours_chemin(g, vus, None, u)
    # s'il n'existe pas de chemin
    if ........
        return None
    # sinon on construit le chemin
    ch = []
    s = v
    while ........
        ch.append(s)
        s = ........
    ch.reverse()
    return ch
```

Exercice 14

Un parcours en profondeur permet de déterminer s'il existe un cycle dans un graphe donné. Si, lors de ce parcours, le sommet qui vient d'être découvert a un voisin en cours de traitement (gris), c'est qu'un cycle existe. Le principe plus en détail est le suivant :

Lorsque l'on visite un sommet `s`,

- S'il est gris, c'est qu'on vient de découvrir un cycle :
- s'il est noir, on ne fait rien ;
- sinon c'est qu'il est blanc, et on procède ainsi :
  1. on colore le sommet `s` en gris ;
  2. on visite tous ses voisins récursivement ;
  3. enfin, on colore le sommet `s` en noir.

Comme on peut le remarquer, les voisins du sommet `s` sont examinés après le moment où le sommet `s` est coloré en gris, et avant le moment où il est colorié en noir. Ainsi, s'il existe un cycle nous ramenant à `s`, on le trouvera comme étant gris et le cycle sera signalé.

Compléter le programme suivant de sorte qu'il renvoie un booléen indiquant s'il y a un cycle dans le graphe *orienté* `g` avec le principe que l'on vient d'énoncer.

```
def parcours_cycle(g, couleur, s):
    '''parcours en profondeur depuis le sommet s'''
    if couleur[s] == 'gris':
        return ........
    if couleur[s] == 'noir':
        return ........
    couleur[s] = 'gris'
    for v in g.voisins(s):
        if ........
            return True
    couleur[s] = 'noir'
    return False

def existe_cycle(g):
    '''détermine la présence d'un cycle dans le graphe g'''
    couleur = {}
    for s in g.sommets():
        couleur[s] = 'blanc'
    for s in g.sommets():
        if ........
            return True
    return False
```

Exercice 15

Reprendre l'exercice 10 mais avec le parcours en largeur.

Exercice 16

Un arbre binaire peut être vu comme un graphe non orienté. Le parcours en profondeur correspond alors au parcours préfixe.

Écrire une fonction `largeur(arb)` qui prend un arbre binaire `arb` en argument et retourne la liste des valeurs de ses nœuds dans un ordre donné par un parcours en largeur.

Pour cela adapter le programme de parcours en largeur du cours.

Exercice 17

Cet exercice reprend l'exercice 13 mais on se propose cette fois d'utiliser un parcours en largeur pour *construire* un chemin entre deux sommets lorsque c'est possible.

On utilise encore une fois deux fonctions :

1. La fonction `parcours_largeur_ch` qui est très semblable à la fonction `parcours_largeur`, l'argument `vus` n'est plus une liste mais un *dictionnaire* qui associe à chaque sommet visité le sommet qui a permis de l'atteindre.
   Compléter le programme ci-dessous à fin qu'il réalise ce parcours.

```
def parcours_largeur_ch(g, s):
    vus = {}
    vus[s] = None
    ...
```

2. La deuxième fonction `chemin(g, u, v)` permet de construire le chemin entre `u` et `v` s'il existe. Pour cela on « remonte » le dictionnaire `vus` obtenu avec `parcours_largeur_ch`.
   Compléter la fonction ci-dessous pour qu'elle réalise cet algorithme.

```
def chemin(g, u, v):
    '''un chemin de u a v, le cas échéant, None sinon'''
    ...
```

Exercice 18

Le parcours en largeur permet de déterminer la *distance* entre deux sommets `s1` et `s2` d'un graphe, c'est à dire le nombre minimal d'arcs à emprunter pour aller de `s1` à `s2`.
La fonction `parcours_distance` est très semblable à la fonction `parcours_largeur`. La liste `vus` est remplacée par un dictionnaire `dist` des distances du sommet `s` aux autres sommets accessibles du graphe `g`. Ce dictionnaire contient initialement le sommet `s` avec une distance `0` et à chaque fois qu'un sommet `v` est découvert depuis le sommet `u`, la distance de `s` à `v` est égale à la distance de `s` à `u` plus 1 pour l'arc que l'on vient d'emprunter.
Compléter la fonction Python ci-dessous de manière à ce qu'elle réalise ce parcours.

```
def parcours_distance(g, s):
    '''dictionnaire des distances entre s et les autres sommets
    accessibles de g'''
    dist = {}
    ........
    file = File()
    file.ajouter(s)
    while not file.est_vide() :
        u = file.consulter()
        for v in g.voisins(u):
            if ........:
                ........
                file.ajouter(v)
        file.retirer()
    return dist
```

Compléter alors la fonction `distance(g, u, v)` qui donne la distance entre les sommets `u` et `v` si un chemin entre ces deux sommets existe et `None` sinon.

```
def distance(g, u, v):
    '''distance de u a v et None si pas de chemin'''
    dist = parcours_distance(g, u)
    if ........
        return None
    else:
        return ........
```

EXERCICE 2 (6 points)

Cet exercice porte sur les graphes.

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 :

```
visités = liste vide des sommets déjà visités

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

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     ...
```

EXERCICE 1 (6 points)

Cet exercice porte sur la programmation objet en Python et les graphes.

Nous avons représenté sous la forme d'un graphe les liens entre cinq différents sites Web :

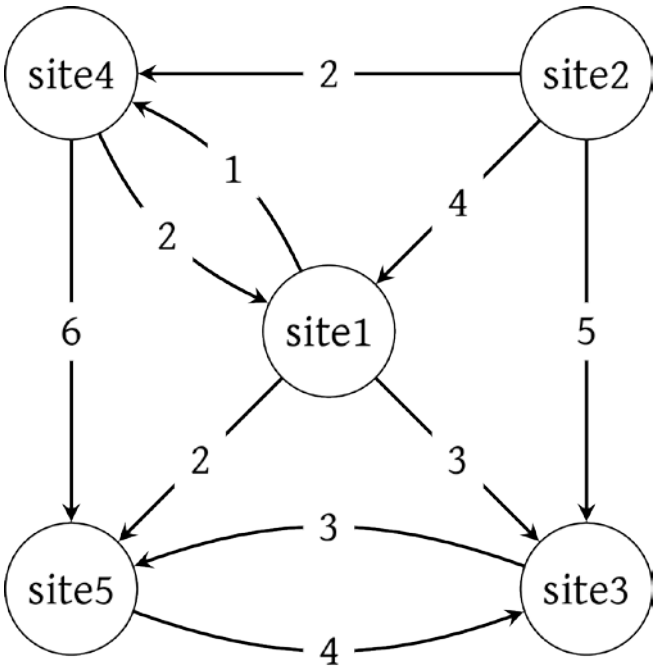


Figure 1. Graphe avec 5 sites

La valeur de chaque arête représente le nombre de citations (de liens hypertextes) d'un site vers un autre. Ainsi, le site **site4** contient 6 liens hypertextes qui renvoient vers le site **site5**.

Les sites sont représentés par des objets de la classe `Site` dont le code est partiellement donné ci-dessous. La complétion de la méthode `calculPopularite` fera l'objet d'une question ultérieure

```
1  class Site:
2
3      def __init__(self, nom):
4          self.nom = nom
5          self.predecesseurs = []
6          self.successeurs = []
7          self.popularite = 0
8          self.couleur = 'blanche'
9
10     def calculPopularite(self):
11         ...
```

Le graphe précédent peut alors être représenté ainsi :

```
1   # Description du graphe
2   s1, s2, s3, s4, s5 = Site('site1'), Site('site2'),
Site('site3'), Site('site4'), Site('site5')
3   s1.successeurs   = [(s3,3), (s4,1), (s5,3)]
4   s2.successeurs   = [(s1,4), (s3,5), (s4,2)]
5   s3.successeurs   = [(s5, 3)]
6   s4.successeurs   = [(s1,2), (s5,6)]
7   s5.successeurs   = [(s3,4)]
8   s1.predecesseurs = [(s2,4), (s4,2)]
9   s2.predecesseurs = []
10  s3.predecesseurs = [(s1,3), (s2,5), (s5,4)]
11  s4.predecesseurs = ...
12  s5.predecesseurs = ...
```

1. Expliquer la ligne 9 de ce code.
2. Les lignes 11 et 12 de cette description du graphe ne sont pas complètes. Recopier et compléter le code des lignes 11 et 12.
3. Donner et expliquer la valeur de l'expression suivante :

```
s2.successeurs[1][1]
```

Pour mesurer la pertinence d'un site, on commence par lui attribuer un nombre appelé valeur de popularité qui correspond au nombre de fois qu'il est cité dans les autres sites, c'est-à-dire le nombre de liens hypertextes qui renvoient sur lui. Par exemple, la valeur de popularité du site **site4** est 3.

4. Donner, selon cette définition, la valeur de popularité du site **site1**.
5. Écrire sur votre copie le code de la méthode `calculPopularite` de la classe `Site` qui affecte à l'attribut `popularite` la valeur de popularité correspondante et renvoie cet attribut.

Afin de calculer cette valeur de popularité pour chacun des sites, nous allons faire un parcours dans le graphe de façon à exécuter la méthode `calculPopularite` pour chacun des objets.

Voici le code de la fonction qui permet le parcours du graphe :

```
1  def parcoursGraphe(sommetDepart):
2      parcours = []
3      sommetDepart.couleur = 'noire'
4      listeS = []
5      listeS.append(sommetDepart)
6      while len(listeS) != 0:
7          site = listeS.pop(0)
8          site.calculPopularite()
9          parcours.append(site)
10         for successeur in site.successeurs:
11             if successeur[0].couleur == 'blanche':
12                 successeur[0].couleur = 'noire'
13                 listeS.append( successeur[0] )
14     return parcours
```

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` ;

- la méthode `pop` enlève de la liste l'élément situé à la position indiquée et le renvoie en valeur de retour;

  par exemple, `tab.pop(2)` enlève l'élément à l'indice 2 et le renvoie.

Dans ce parcours, les sites non encore traités sont de couleur `'blanche'` (valeur par défaut à la création de l'objet) et ceux qui sont traités de couleur `'noire'`.

6. Dans ce parcours, on manipule la liste Python nommée `listeS` uniquement à l'aide d'appels de la forme `listeS.append(sommet)` et `listeS.pop(0)`. Donner la structure de données correspondant à ces manipulations.

7. Donner le nom de ce parcours de graphe.

8. La fonction `parcoursGraphe` renvoie une liste `parcours`. Indiquer la valeur renvoyée par l'appel de fonction :

   ```
   parcoursGraphe(s1)
   ```

On cherche maintenant le site le plus populaire, celui dont la valeur de popularité est la plus grande.

Voici le code de la fonction qui renvoie le site le plus populaire, elle prend comme argument une liste non vide contenant des instances de la classe `Site`.

```
1  def lePlusPopulaire(listeSites):
2      maxPopularite = 0
3      siteLePlusPopulaire=listeSites[0]
4      for site in listeSites:
5          if site.popularite > maxPopularite:
6              ...
7              ...
8      return siteLePlusPopulaire
```

9. Copier et compléter les lignes 6 et 7 de cette fonction.

10. Donner ce que renvoie la ligne de code suivante :

```
lePlusPopulaire(parcoursGraphe(s1)).nom
```

11. On envisage d'utiliser l'ensemble des fonctions proposées ci-dessus pour rechercher le site le plus populaire parmi un très grand nombre de sites (quelques milliers de sites). Expliquer si ce code est adapté à une telle quantité de sites à traiter. Justifier votre réponse.