Définitions

- Un **graphe non orienté** est un couple $G = (V, E)$ où V est un ensemble fini de **sommets** et E est un ensemble d'**arêtes**, chaque arête étant un ensemble de deux sommets de V.
  Si $e = \{u, v\} \in E$, on dit que u et v sont **adjacents** ou **voisins** et que ce sont les extrémités de e.
  Le **degré** $\deg(v)$ d'un sommet v est son nombre de voisins.
  Une **feuille** est un sommet de degré 1.

- Un **graphe orienté** est la même chose qu'un graphe non orienté, sauf que chaque arête est un couple (u, v) (représenté par $u \longrightarrow v$) au lieu d'un ensemble.

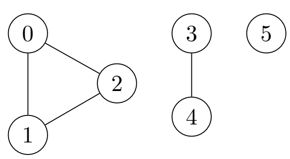
- Si G est un graphe non orienté à n sommets et p arêtes alors

$$p \leq \binom{n}{2} = O(n^2)$$

  En effet, il y a $\binom{n}{2}$ arêtes possibles (il faut choisir les deux extrémités pour avoir une arête). De plus, un graphe avec toutes les arêtes possibles ($p = \binom{n}{2}$) est dit **complet**.

Représentations

- Exemple de graphe avec ses représentations en Python :



Matrice d'adjacence

```
[[0, 1, 1, 0, 0, 0],
 [1, 0, 1, 0, 0, 0],
 [1, 1, 0, 0, 0, 0],
 [0, 0, 0, 0, 1, 0],
 [0, 0, 0, 1, 0, 0],
 [0, 0, 0, 0, 0, 0]]
```

Liste d'adjacence

```
[[1, 2],
 [0, 2],
 [0, 1],
 [4],
 [3],
 []]
```

  Une matrice adjacence d'un graphe non orienté est toujours symétrique.
  Dans le cas d'un graphe pondéré : on met le poids de l'arête au lieu de 1 pour une matrice d'adjacence.

- Afficher les voisins d'un sommet `u` dans un graphe G :

Matrice d'adjacence

```python
for j in range(len(G[u])):
    if G[u][v] == 1:
        print(v)
```

Liste d'adjacence

```python
for v in G[u]:
    print(v)
```

- Exercice : Écrire deux fonctions, pour convertir une matrice d'adjacence en liste d'adjacence et inversement.
  Solution :

```python
def mat_to_list(G):
    n = len(G)
    L = [[] for _ in range(n)]
    for u in range(n):
        for v in range(n):
            if G[u][v] == 1:
                L[u].append(v)
    return L

def list_to_mat(G):
    n = len(G)
    M = [[0 for _ in range(n)] for _ in range(n)]
    for u in range(n):
        for v in G[u]:
            M[u][v] = 1
    return M
```

- Exercice : Écrire une fonction pour inverser le sens de toutes les arêtes d'un graphe orienté représenté par matrice d'adjacence.
  Solution : Il s'agit de la matrice transposée.

```python
def inverse(G):
    n = len(G)
    R = [[0] * n for _ in range(n)]
    for u in range(n):
        for v in range(n):
            R[v][u] = G[u][v]
    return R
```

- Pour un graphe orienté à n sommets et p arêtes :

	Matrice d'adjacence	Liste d'adjacence
complexité mémoire	$O(n^2)$	$\boxed{O(n+p)}$
tester si $\{u, v\} \in E$	$\boxed{O(1)}$	$O(\deg(u))$
parcourir voisins de u	$O(n)$	$\boxed{O(\deg(u))}$

Connexité, cycle

- Un **chemin** dans $G = (V, E)$ est une suite de sommets $u_0, u_1, \ldots, u_n$ tels que $\{u_i, u_{i+1}\} \in E$, $\forall i \in \{0, \ldots, n-1\}$.
  Un chemin est un **cycle** s'il revient au sommet de départ ($u_0 = u_n$).

- Un graphe est **acyclique** s'il ne contient pas de cycle.

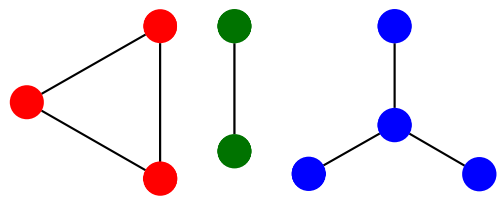
- Un graphe G non orienté est **connexe** si, pour tout sommets u et v, il existe un chemin entre u et v dans G.

- La relation suivante est une relation d'équivalence sur un graphe non orienté :

$$u \sim v \iff \text{il existe un chemin entre } u \text{ et } v$$

  Les classes d'équivalences pour $\sim$ sont les sous-graphes connexes maximaux (au sens de $\subseteq$) de G, ils sont appelés **composantes connexes**.

Exemple : un graphe avec 3 composantes connexes.



Remarque : Un graphe est connexe ssi il contient une unique composante connexe.

Parcours de graphe

- **Parcours en profondeur** : on visite les sommets le plus profondément possible avant de revenir en arrière.

```python
def dfs(G, s): # G est représenté par liste d'adjacence
    visited = [False]*len(G)
    def aux(u):
        if not visited[u]:
            # traiter u (l'afficher par exemple)
            visited[u] = True
            for v in G[u]:
                aux(v)
    aux(s)
```

Complexité avec représentation par liste d'adjacence : Soit n le nombre de sommets et p le nombre d'arêtes.
`[False]*len(G)` est en $\mathrm{O}(n)$.
Chaque arête est parcourue au plus une fois, d'où $\mathrm{O}(p)$ appels récursifs de `aux`.
Au total : $\boxed{\mathrm{O}(n+p)}$.
Remarque : La complexité est $\mathrm{O}(n^2)$ avec une matrice d'adjacence.

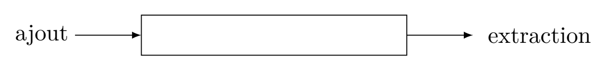
- Application : déterminer si un graphe est connexe, en regardant si `visited` ne contient que des `True` à la fin du parcours.

```python
def connexe(G):
    visited = [False]*len(G)
    def aux(u):
        if not visited[u]:
            visited[u] = True
            for v in G[u]:
                aux(v)
    aux(0)
    for b in visited: # ou : return all(visited)
        if not b:
            return False
    return True
```

- Application : détection de cycle. Pour cela, on peut tester si on revient sur un sommet déjà visité, à condition de ne pas revenir sur le sommet parent (car parcourir une arête dans les deux sens n'est pas considéré comme un cycle).

```python
def has_cycle(G, s):
    # renvoie True si G a un cycle atteignable depuis s
    visited = [False]*len(G)
    def aux(u, p): # p : sommet ayant permis de découvrir u
        if not visited[u]:
            visited[u] = True
            for v in G[u]:
                if v != p and aux(v, u):
                    return True
        return False
    return aux(s, -1)
```

- Une **file** est une structure de donnée possédant trois opérations :
  – Ajout d'un élément à la fin de la file.
  – Extraction (suppression et renvoi) de l'élément au début de file. Ainsi, l'élément extrait est l'élément le plus ancien.
  – Test pour savoir si la file est vide.



La classe `deque` du module `collections` de Python implémente une file, où l'ajout est réalisée par `appendleft` et l'extraction par `pop` :

```python
from collections import deque

q = deque() # file vide
q.appendleft(4)
q.appendleft(7)
q.pop() # renvoie 4
q.appendleft(-5)
q.pop() # renvoie 7
```

- **Parcours en largeur** : on visite les sommets par distance croissante depuis le sommet de départ (le sommet de départ, puis les voisins, puis les voisins des voisins...). Pour cela, on stocke les prochains sommets à visiter dans une file `q` :

```python
def bfs(G, s):
    visited = [False]*len(G)
    q = deque([s])
    while len(q) > 0:
        u = q.pop()
        if not visited[u]:
            visited[u] = True
            for v in G[u]:
                q.appendleft(v)
```

- Application : calcul de distance (en nombre d'arêtes), en stockant des couples (sommet, distance) dans `q`.

```python
def distances(G, s):
    dist = [-1]*len(G)
    q = deque([(s, 0)])
    while len(q) > 0:
        u, d = q.pop()
        if dist[u] == -1:
            dist[u] = d
            for v in G[u]:
                q.appendleft((v, d + 1))
    return dist
```