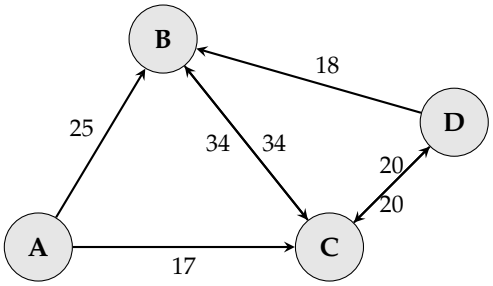
Parcours de graphes

Structures de données

1 Implémentations

1.1 Matrice d'adjacence

On indique la présence ou le poids d'un arc à l'intersection de la ligne (sommet de départ) et de la colonne (sommet d'arrivée).

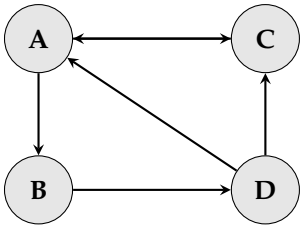


$$A = \begin{pmatrix} 0 & 25 & 17 & 0 \\ 0 & 0 & 34 & 0 \\ 0 & 34 & 0 & 20 \\ 0 & 18 & 20 & 0 \end{pmatrix}$$

Remarque. Il faut un tableau de n^2 entiers pour stocker un graphe de n sommets. Dans le cas d'un graphe non orienté, la matrice est symétrique.

1.2 Liste d'adjacence

Pour chaque sommet, on mémorise la liste de ses voisins.

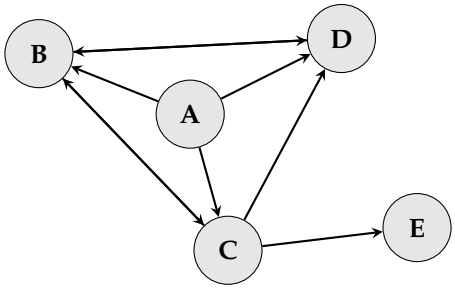



```
[[1, 2], [3], [0], [0, 2]]
```

avec A:0, B:1, C:2, D:3.

1.3 Dictionnaire d'adjacence

L'étiquette d'un sommet sert de clé et on lui associe la liste de ses voisins :

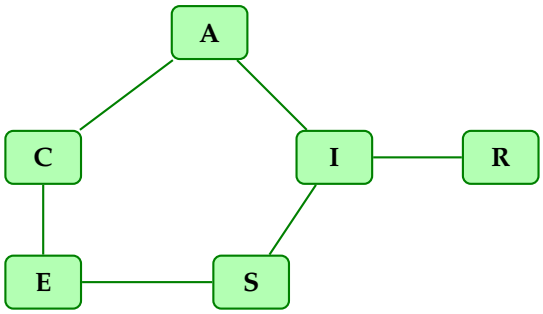



```
{"A": ["B","C","D"], "B": ["C","D"],
 "C": ["B","D","E"], "D": ["B"], "E": []}
```

Par la suite, nous travaillerons sur des graphes **non orientés connexes**, représentés par dictionnaire.

② Parcours

On illustre les parcours sur le graphe suivant :




```
G = {
  "A": ["C", "I"],
  "C": ["A", "E"],
  "I": ["A", "R", "S"],
  "R": ["I"],
  "E": ["C", "S"],
  "S": ["E", "I"]
}
```

2.1 Parcours en profondeur (DFS)

Le parcours en profondeur avance le plus loin possible avant de revenir en arrière. Il utilise une **pile** (explicite ou via la récursivité).

2.1.1 Version récursive

```
def parcourir_profondeur_recursif(G, S, deja_vus):
    deja_vus.append(S)          # marquer S
    print(S, end=" ")           # traiter S
    for voisin in G[S]:
        if voisin not in deja_vus:
            parcourir_profondeur_recursif(G, voisin, deja_vus)

parcourir_profondeur_recursif(G, "A", [])
```

Résultat : `A C E S I R`.

Le parcours passe par le **premier** voisin non marqué : A → C (premier voisin de A) → E → S → I → R.

2.1.2 *Version itérative*

```
def parcourir_profondeur_iteratif(G, S, deja_vus):
    pile_a_traiter = [S]
    while pile_a_traiter != []:
        S = pile_a_traiter.pop()       # dépiler
        if S not in deja_vus:
            deja_vus.append(S)         # marquer
            print(S, end=" ")          # traiter
            for voisin in G[S]:
                if voisin not in deja_vus:
                    pile_a_traiter.append(voisin)

parcourir_profondeur_iteratif(G, "A", [])
```

Résultat : `A I S E C R`.

Le parcours passe par le **dernier** voisin empilé (sommet de la pile) : A → I → S → E → C → R.

2.2 Parcours en largeur (BFS)

Le parcours en largeur examine les voisins directs, puis les voisins des voisins, etc. Il utilise une **file**.

```
def parcourir_largeur(G, S, deja_vus):
    file_a_traiter = [S]
    deja_vus.append(S)              # marquer S
    while file_a_traiter != []:
        S = file_a_traiter.pop(0)   # défiler
        print(S, end=" ")           # traiter
        for voisin in G[S]:
            if voisin not in deja_vus:
                deja_vus.append(voisin)
                file_a_traiter.append(voisin)

parcourir_largeur(G, "A", [])
```

Résultat : `A C I E R S` (A, puis ses voisins C et I, puis les voisins à distance 2).

2.3 Comparaison des deux parcours

Les deux parcours itératifs ont une structure quasi identique. Deux points les distinguent :

	DFS (profondeur)	**BFS (largeur)**
Structure	**Pile** (LIFO)	**File** (FIFO)
Marquage	Au **dépilement**	À l'**enfilement**

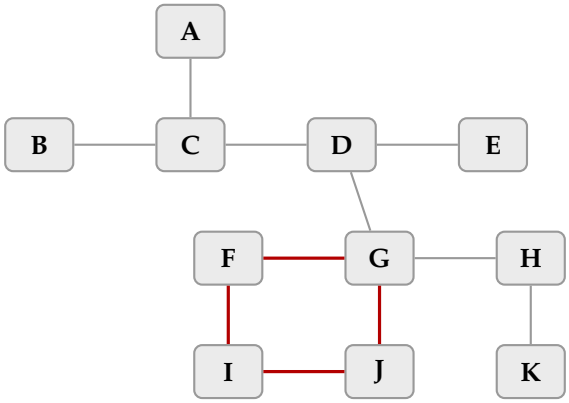
Pourquoi marquer au dépilement en DFS ? Un même sommet peut être empilé plusieurs fois par des voisins différents. On ne sait pas encore par quelle arête le parcours y arrivera.

Pourquoi marquer à l'enfilement en BFS ? Le BFS explore par distance croissante ; quand on enfile un voisin, on sait que c'est par le chemin le plus court en nombre d'arêtes. Marquer évite les doublons.

Propriété du BFS. Le parcours en largeur permet de trouver un **plus court chemin en nombre d'arêtes** entre deux sommets.

③ Détection de cycle

Un **cycle** est détecté lorsqu'on rencontre un sommet déjà marqué lors du parcours en profondeur.



Le cycle G–F–I–J–G est visible (arêtes en rouge).

```
def possede_cycle(G, S, deja_vus):
    pile_a_traiter = [S]
    while pile_a_traiter != []:
        S = pile_a_traiter.pop()
        if S in deja_vus:       # déjà parcouru ?
            return True         # cycle détecté !
        deja_vus.append(S)
        for voisin in G[S]:
            if voisin not in deja_vus:
                pile_a_traiter.append(voisin)
    return False
```

La présence de doublons dans la pile se traduit par la présence d'un cycle : la première occurrence est marquée lors de son dépilement ; c'est lors du dépilement de la seconde (arrivée par une arête différente) qu'on constate le marquage et donc le cycle.

④ Recherche d'un chemin

On utilise le BFS en mémorisant le **prédécesseur** de chaque sommet visité dans un dictionnaire, puis on reconstitue le chemin en remontant du sommet d'arrivée au départ.

```
def recherche_chemin(G, S, D):
    M = {}                      # les prédécesseurs
    R = []                      # le chemin inversé
    if parcourir(G, S, D, M):
        R.append(D)
        while D != S:
            D = M[D]
            R.append(D)
```

```
    R.reverse()
    return R

def parcourir(G, S, D, deja_vus):
    file_a_traiter = [S]
    deja_vus[S] = None
    while file_a_traiter != []:
        S = file_a_traiter.pop(0)   # défiler
        if S == D:                  # destination atteinte
            return True
        for voisin in G[S]:
            if voisin not in deja_vus:
                deja_vus[voisin] = S    # prédécesseur
                file_a_traiter.append(voisin)
    return False

>>> recherche_chemin(G, "A", "K")
['A', 'C', 'D', 'G', 'H', 'K']
```