

## Implémenter les graphes en Python

**Exercice 1** (★). 1. Créer une fonction `CreerDico(L)` qui à une liste `L` va renvoyer un dictionnaire `D` tel que `L[i]` soit une clé et dont la valeur est `i`. Ainsi si `L=["Luxembourg","Port-Royal","GDN"]`, alors `CreerDico` renvoie le dictionnaire `D={"Luxembourg":0,"Port-Royal":1,"GDN":2}`.

2. Créer une fonction `CreerGraphe(L)` qui va renvoyer un graphe dont les sommets sont les éléments de `L` et qui n'a initialement aucune arête. Ce graphe sera modélisé par un couple `(D,M)` où `D` est un dictionnaire associant à chaque sommet un numéro, et `M` sera la matrice nulle (créée comme une liste de listes).

3. Utiliser cette fonction avec la commande `G=CreerGraphe(L)` où `L` est la liste utilisée à la question 1. Ainsi, `G` est un graphe où, pour l'instant, aucun sommet n'est relié à un autre sommet.

On rappelle que lorsque que l'on modélise un graphe non pondéré par une matrice `M`, `M[i][j]` vaut 1 s'il existe un arc partant du sommet de numéro `i` et allant vers le sommet de numéro `j`. On rappelle aussi que les listes subissent un **effet de bord** : ils seront modifiés au sein de la fonction et cette modification sera effective même sans `return`.

4. Créer une fonction `RajouterArete(G,s,b)` qui va rajouter un arc dans le graphe `G` partant du sommet `s` et allant vers le sommet `b` pour cela, il suffira de modifier la matrice du graphe pour que le coefficient concerné vaille 1. Attention `s` et `b` sont des noms de sommets et non les numéros de ces sommets. Ainsi, pour rajouter une arête entre Luxembourg et Port-Royal, on écrira `RajouterArete(G,"Luxembourg","Port-Royal")` ce qui est plus pratique que `RajouterArete(G,0,1)`.

Comme pour les listes, on peut boucler sur les clés d'un dictionnaire :

```
for k in D:#k va prendre successivement toutes les clés du dico
    print(k,D[k])          #k clé, D[k] valeur de la clé associée
```

5. Créer une fonction `ListeVoisins(G,s)` qui a un graphe `G` et un sommet `s` va envoyer la liste des voisins de `s` c'est-à-dire les sommets `v` tels qu'il existe un arc partant de `s` et allant vers `v`.

## Implémentation des piles et des files en Python

Le code suivant<sup>1</sup> permet de créer des piles et les files en Python avec `deque`, ainsi que d'enlever ou d'ajouter des éléments.

<sup>1</sup>. Commandes non exigibles comme toutes celles provenant des bibliothèques comme `numpy`, `matplotlib.pyplot` ou `time`.

```
from collections import deque#import de deque
```

```
file=deque([8,2,"Paris","Lyon"])#transformer une liste en deque
file.append("Marseille")#on rajoute un élément à la file
file.popleft()#On enlève le premier arrivé
```

```
pile=deque([1,2,"Paris","Lyon"])
pile.append('32')
enleve=pile.pop()#On enlève le dernier arrivé
```

**Exercice 2** (★). Comparer le temps mis pour retirer un élément d'une liste par `pop(0)` par `pop()` avec le temps mis en utilisant les piles et les files générées par `deque`. Pour cela, il suffit de créer une pile avec une liste, une file avec une liste, une pile avec `deque` et une file avec `deque` toutes de longueur `N` et de mesurer le temps mis pour leur retirer le premier/dernier élément dans chacun de ces quatre cas.

## Sortir d'un labyrinthe

**Exercice 3** (★★ YT). On considère un labyrinthe modélisé comme une matrice (une liste de listes) : les cases blanches sont codées comme des 1 et les cases noires sont codées comme des 0. On va coder un parcours en largeur pour sortir du labyrinthe partant de l'entrée.

1. Téléchargez les fichiers `TP12M1.txt` et `TP12M2.txt` qui sont sur CDP et mettez les dans le même dossier que votre script Python actuel.

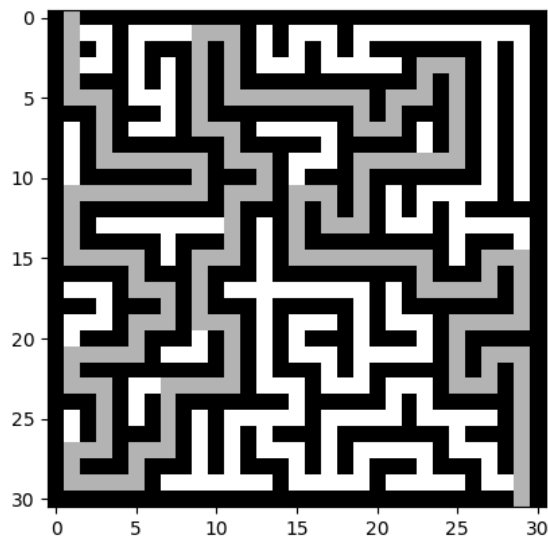
2. Les commandes suivantes ouvrent le labyrinthe :

```
import matplotlib.pyplot as plt#bibliothèque pour les images
import time as time#bibliothèque pour mesurer le temps
import pickle#bibliothèque pour importer des fichier
```

```
with open("TP12M1.txt", "rb") as fichier:
    M=pickle.load(fichier)#M: matrice chargée à partir d'un fichier
print(M[0][0],M[0][1],M[0][2])
plt.figure()
plt.imshow(M,cmap="gray")#On affiche la matrice en niveau de gris
plt.show()
n,p=len(M),len(M[0])#n: le nombre de lignes, p: le nombre de colonnes
source=(0,1)#entrée du labyrinthe
but=(n-1,p-2)#sortie du labyrinthe
```

Attention, ici `M` n'est pas une matrice d'adjacence. Elle indique juste la couleur des pixels (mur ou pas mur).

- Créer une fonction `ListeVoisins(M,sommet)` qui renvoie la liste des voisins du sommet (modélisé comme un couple d'entiers), c'est-à-dire la liste des **cases blanches** qui sont à côté (une case en haut, une case en bas, une case à droite ou une case à gauche de `sommet`).
- Coder le parcours en largeur partant du sommet `source` en codant une fonction `ParcoursLargeur(M,source)`
- Modifier la fonction précédente, pour indiquer à chaque fois que l'on visite un sommet par quel sommet on est passé pour le visiter. Cela consiste donc à créer un dictionnaire `P` (dictionnaire des prédécesseurs) tel que `P[v]=sommet` si le sommet `v` a été ajouté à la file en tant que voisin de `sommet`.
- Tracer le chemin de l'entrée (sommet `source`) à la sortie (sommet `but`), pour cela partir du sommet `but` et remonter via le dictionnaire des prédécesseurs, chacun de ces sommets sera modifié dans la matrice, le coefficient ne vaudra plus 1 (case blanche) mais 0.5 (case grise).
- Vérifier si un élément est déjà dans la liste des sommets visités à une complexité proportionnelle au nombre d'éléments de la liste, tandis que vérifier si un élément est une clé d'un dictionnaire a un coût constant. Écrire donc une nouvelle version du parcours en largeur avec une meilleure complexité. Comparer les temps nécessaires avec les deux labyrinthes : TP12M1 et TP12M2 et les deux versions du parcours en largeur.



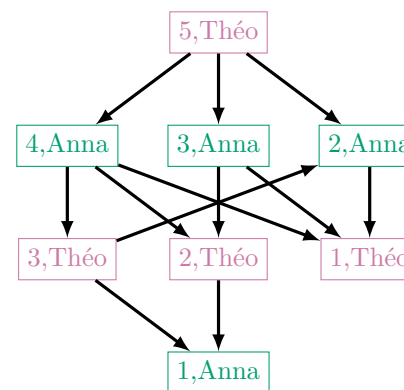
## Créer un graphe par un parcours : le jeu de Nim

**Exercice 4 (\*\*).** Imaginons deux joueurs et un tas de  $N$  allumettes. Chacun à son tour, les joueurs jouent en respectant deux règles :

- Ils enlèvent une, deux ou trois allumettes au choix
- Mais ils doivent laisser au moins une allumette dans le tas.

Celui qui doit jouer alors qu'il reste exactement une allumette a perdu car il ne peut pas jouer en respectant les deux règles :

Prenons un exemple, où Théo et Anna jouent avec 5 allumettes. Théo commence. On peut faire un graphe de toutes les possibilités :



Le sommet 5,Théo veut dire qu'il y a actuellement cinq allumettes et que c'est à Théo de jouer. Le sommet 1,Anna veut dire qu'il reste 1 allumette et que c'est à Anna de jouer, elle a donc perdu. De même 1,Théo veut dire que Théo a perdu. Ainsi, une des parties possibles serait :

$$5, \text{Théo} \xrightarrow{\text{Théo}} 4, \text{Anna} \xrightarrow{\text{Anna}} 2, \text{Théo} \xrightarrow{\text{Théo}} 1, \text{Anna}$$

Ainsi, Anna perd cette partie. Cherchons maintenant à implémenter ce graphe en Python. en partant d'un sommet de la forme  $(n, j)$  on va aller visiter les trois successeurs de ce sommet :  $(n-1, o), (n-2, o), (n-3, o)$  où  $o$  est le nom de l'opposant du joueur, mais attention, il faut garder en tête que le nombre d'allumettes doit être supérieure ou égale à 1. Coder la création de ce graphe avec un dictionnaire d'adjacence on utilisera un parcours récursif puis avec un parcours en largeur avec une file puis avec un parcours en profondeur avec une pile.