# Parcours de graphe

## Rappels

Dans cette partie, nous utiliserons le principe des piles en utilisant pour les listes, les méthodes `.pop()` et `.append()`.

— La méthode `.pop()` prélève le dernier élément de la liste. On dit alors que l'on **dépile** :

```
1  L = [1,2,4]
2  >>>L.pop()
3  4
4  >>>L
5  [1,2]
```

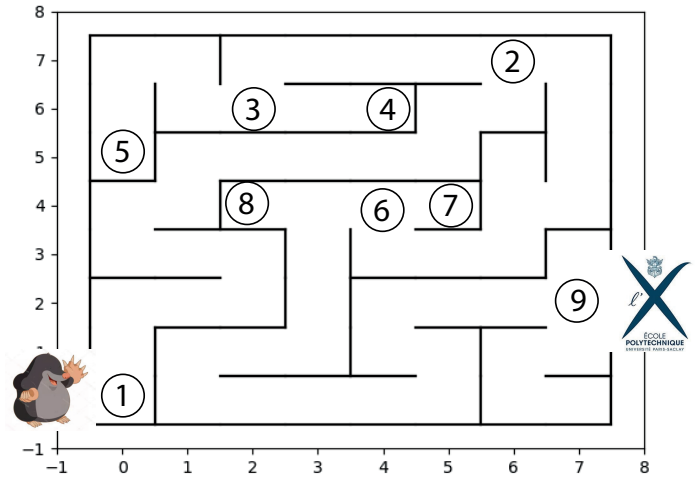
— la méthode `.append(elt)` permet de rajouter un élément sans changer l'adresse de la liste. On dit alors que l'on **empile** :

```
1  L = [1,2,4]
2  >>>L.append(6)
3  >>>L
4  [1,2,4,6]
```

## 1 - Sortir d'un labyrinthe

### a) description par graphe orienté

On considère le labyrinthe suivant. Le but est d'aider René à atteindre la sortie.




**Q 1 -** Représenter sur une feuille de brouillon le graphe correspondant. Les principaux points d'intersection sont indiquées par des numéros et seront les sommets de votre graphe.

**Q 2 -** Décrire ce graphe **orienté** dans le sens de parcours sous la forme d'un dictionnaire `labyrinthe`.

### b) algorithme de parcours en profondeur

Pour dénombrer tous les chemins possibles à partir du sommet `i`, l'algorithme est le suivant :

— on initialise une liste `chemins_temp` contenant la sous-liste d'un seul élément `[i]` . `chemins_temp` est une liste temporaire qui contiendra les différents chemins en cours de construction.

— On initialise une liste `chemins` vide qui contiendra la sauvegarde des chemins possible

— tant que `chemins_temp` est non vide :

  — on extrait le dernier chemin de la liste (utiliser la méthode `.pop()`.

— on regarde les successeurs du dernier sommet visité
  — si ils existent, on crée autant de chemins qu'il y a de successeurs en ajoutant ces sommets au chemin en cours, puis on les stocke dans `chemins_temp` ;
  — s'il n'y a pas de successeurs, on sauvegarde le chemin dans la liste `chemins`

Pour fixer les idées, partons du chemin [1,2]. Les successeurs de 2 sont 3 et 6. On crée alors deux sous listes [1,2,3] et [1,2,6] qui sont stockées dans `chemins_temp`.

**Q 3 -** Proposer une fonction `ajoute_sommet(dico,chemin` qui, à partir d'une liste `chemin` possible, renvoie autant de chemins qu'il y a de successeurs au dernier sommet de `chemin`. La fonction devra valider les tests suivants :

```
>>> ajoute_sommet(labyrinthe,[1,2])
[1,2,3], [1,2,6]
>>> ajoute_sommet(labyrinthe,[1,2,3,4])
[]
```

**Q 4 -** En traduisant l'algorithme ci-dessus, proposer une fonction `parcours(dico,i=1)` qui renvoie tous les chemins possibles à partir du sommet `i`. Votre fonction devra vérifier :

```
>>> parcours(labyrinthe)
[ [1,2,3,4],[1,2,3,5], [1,2,6,8],[1,2,6,9],[1,2,6,7] ]
```

**Q 5 -** Proposer une fonction `sortie(dico,i=1,j=9)` qui renvoie le chemin vers la sortie. Votre fonction devra vérifier :

```
>>> sortie(labyrinthe)
[1,2,6,9]
```

## 2 - Cas d'un graphe non-orienté

### a) Graphe non pondéré

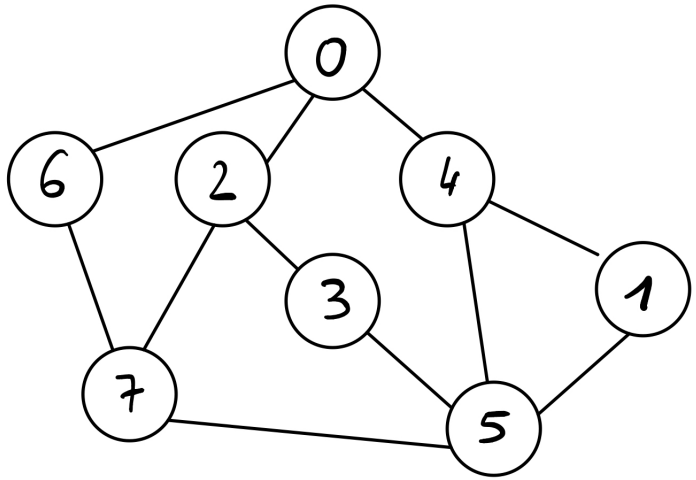
On se base sur le graphe suivant déjà utilisé dans le TP précédent.

On reprendra le TP précédent où ce graphe est décrit en terme de dictionnaire. On cherche à obtenir tous les chemins possibles pour aller d'Amiens (0) à Fréjus (5) sans repasser deux fois par la même ville.

**Q 6 -** Modifier la fonction `ajoute_sommet` de telle sorte qu'elle renvoie autant de chemins qu'il y a de successeurs **non encore présent dans la liste** au dernier sommet de chemin.

**Q 7 -** Ecrire une fonction `chemins_possibles(i,j)` qui reprend la trame des algorithmes précédents et qui renvoie la liste des chemins possibles pour aller de `i` à `j`.

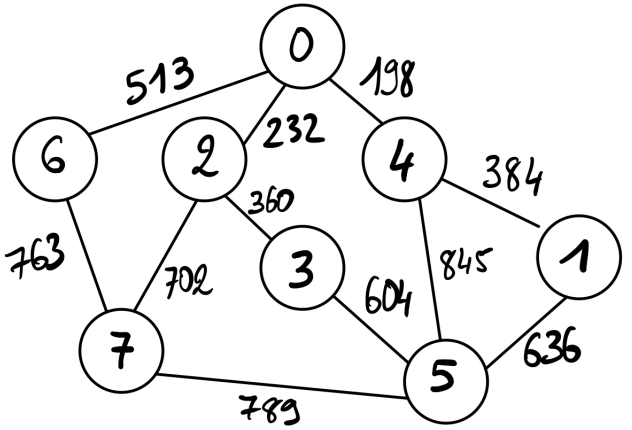
On s'attend à :




```
>>> chemins_possibles(0,5)
[[0,6,7,5],[0,2,3,5],[0,4,5],[0,4,1,5],[0,6,7,2,3,5],[0,2,7,5] ]
```

### b) Graphe pondéré

On repart du graphe de départ non orienté que l'on pondère par la distance entre les différentes villes. On obtient le graphe suivant :




**Q 8 -** Proposer une fonction `distance(dico,chemin)` qui retourne la distance parcourue en utilisant le `chemin` valide.

**Q 9 -** Conclure en proposant une fonction `plus_court_chemin(dico,i,j)` qui renvoie le plus court chemin reliant `i` à `j`.