Parcours d'un labyrinthe

Rappels

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

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

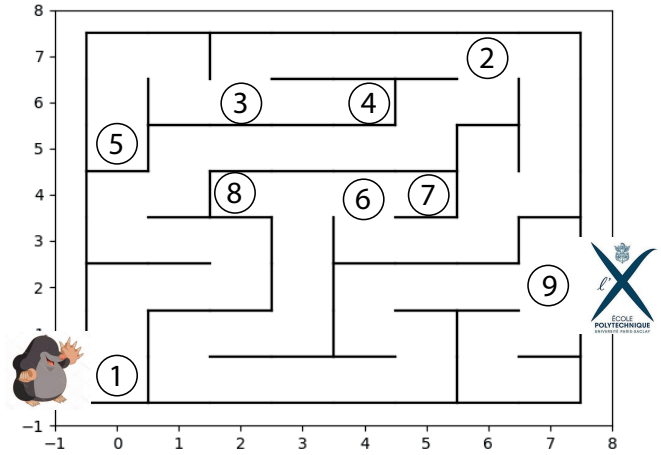
```
L = [1,2,4]
>>>L.pop()
4    # L = [1,2]
```

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

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

1 - Description par Dictionnaire

On considère le labyrinthe représenté par la figure suivante. Le but est d'aider René à atteindre la sortie en partant du sommet (1).



Q 1 - Les principaux points d'intersection sont indiquées par des numéros et seront les sommets de votre graphe. Décrire ce graphe orienté selon les numéros croissants sous la forme d'un dictionnaire `labyrinthe`. On considérera qu'un sommet ne peut pointer que vers des numéros supérieurs au sien pour éviter les retours en arrière. Un sommet pointant vers une seule valeur sera écrit tout de même écrit sous la forme d'un *singleton* ou liste d'un seul élément. Par exemple : `1 : [2]` .

2 - Chemins possibles

Q 2 - 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`.

D'après le schéma du labyrinthe, à partir du `chemin = [1,2]`, il existe deux chemins possibles `[1,2,3] et [1,2,6]` . En revanche, s'il n'existe pas de possibilité de continuer, la fonction doit renvoyer une liste vide []. 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])
[]
```

3 - Parcours vers la sortie

Pour parvenir à la sortie du labyrinthe, on effectue un parcours en profondeur en utilisant l'algorithme suivant en partant du sommet i :

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

— tant que `chemins` est non vide :

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

    — Si la sortie j appartient au chemin, on retourne le chemin correspondant ;

    — s'il n'y a pas possibilité de continuer, `chemins` n'est pas modifié ;

    — sinon, on ajoute à `chemins` les différentes possibilités existantes.

Q 3 - 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]
```

AM, JB, NG, SR

Q 4 - Justifier le résultat suivant :

```
>>> sortie(labyrinthe,5,9)
None
```

4 - Cas d'un graphe non-orienté

Q 5 - Décrire le graphe précédent sans orientation sous la forme d'un dictionnaire qui commencera par : {1:[2],2:[1,3,6],...} .

Q 6 - Proposer une nouvelle 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` sans retour en arrière. La fonction devra valider les tests suivants :

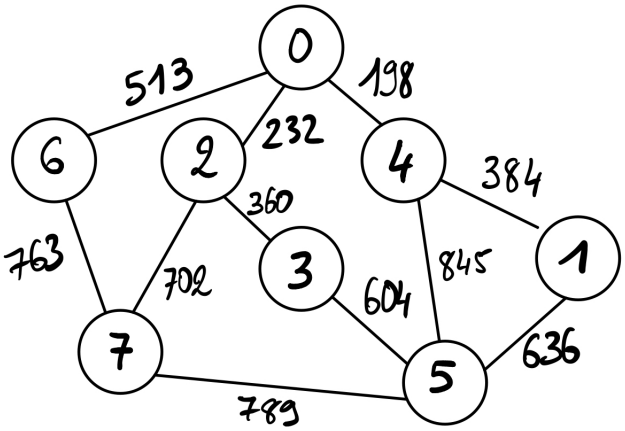
```
>>> ajoute_sommet(labyrinthe,[1,2])
[1,2,3], [1,2,6]
>>> ajoute_sommet(labyrinthe,[1,2,3,4])
[]
```

Q 7 - Vérifier que votre algorithme de sortie donne bien :

```
>>> sortie(labyrinthe,5,9)
[5,3,2,6,9]
```

5 - Graphe pondéré

On reprendra le TP précédent où le 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 8 - Définir une variable `dico_pond` représentant ce graphe. Chaque clé est associée à un sommet. La valeur de la clé est un dictionnaire des sommets adjacents, dictionnaire dont les clés sont les sommets adjacents et les valeurs la distance entre le sommet de départ et le sommet adjacent. Le début du dictionnaire est donc :

dico_pond = {0:{2:232, 4:198, 6:513}, ... }

On peut alors accéder aux distances entre deux sommets adjacents i et j à l'aide de la syntaxe `dico_pond[i][j]`

Q 9 - Proposer une fonction `distance(dico,chemin)` qui retourne la distance parcourue en utilisant le `chemin` (supposé valide).

Q 10 - 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] ]
```

Q 11 - Conclure en proposant une fonction `plus_court_chemin(dico,i,j)` qui renvoie le plus court chemin reliant `i` à `j`.