BAC BLANC 2022 - SUJET 3
mercredi 26/01/2022
<RATTRAPAGE>

ÉPREUVE D'ENSEIGNEMENT DE SPÉCIALITÉ

JANVIER 2022

NUMÉRIQUE ET SCIENCES INFORMATIQUES

Durée de l'épreuve : **3 heures 30**

L'usage de la calculatrice et du dictionnaire n'est pas autorisé.

Dès que ce sujet vous est remis, assurez-vous qu'il est complet.

Ce sujet comporte 14 pages numérotées de 1/14 à 14 /14.

Le candidat traite au choix 3 exercices
parmi les 5 exercices proposés

Chaque exercice est noté sur 4 points.

EXERCICE 1 (4 points)

Principaux thèmes abordés : structure de données (tableaux, dictionnaires) et langages et programmation (spécification).

Objectif de l'exercice :

Les Aventuriers du Rail© est un jeu de société dans lequel les joueurs doivent construire des lignes de chemin de fer entre différentes villes d'un pays.

La carte des liaisons possibles dans la région Occitanie est donnée en **annexe 1 de l'exercice 1**. Dans l'**annexe 2 de l'exercice 1**, les liaisons possédées par le joueur 1 sont en noir, et celles du joueur 2 en blanc. Les liaisons en gris sont encore en jeu.

Codages des structures de données utilisées :

- **Liste des liaisons d'un joueur :** Toutes les liaisons directes (sans ville intermédiaire) construites par un joueur seront enregistrées dans une variable de type "**tableau de tableaux**".

  Le joueur 1 possède les lignes directes **"**Toulouse-Muret", "Toulouse-Montauban", "Gaillac-St Sulpice" et "Muret-Pamiers" (liaisons indiquées en noir dans l'**annexe 2 de l'exercice 2**). Ces liaisons sont mémorisées dans la variable ci-contre.

```
liaisonsJoueur1 = [
 ["Toulouse","Muret"],
 ["Toulouse","Montauban"],
 ["Gaillac","St Sulpice"],
 ["Muret","Pamiers"]
]
```

  Remarque : *Seules les liaisons directes existent, par exemple* ***["Toulouse","Muret"]*** *ou* ***["Muret","Toulouse"]****. Par contre, le tableau* ***["Toulouse","Mazamet"]*** *n'existe pas, puisque la ligne Toulouse-Mazamet passe par Castres.*

- **Dictionnaire associé à un joueur :** On code la liste des villes et des trajets possédée par un joueur en utilisant un **dictionnaire de tableaux**. Chaque **clef de ce dictionnaire** est une ville de départ, et chaque **valeur** est un **tableau** contenant les villes d'arrivée possibles en fonction des liaisons possédées par le joueur.

  Le **dictionnaire de tableaux** du joueur 1 est donné ci-contre :

```
DictJoueur1 = {
    "Toulouse":["Muret","Montauban"],
    "Montauban":["Toulouse"],
    "Gaillac":["St Sulpice"],
    "St Sulpice":["Gaillac"],
    "Muret":["Toulouse","Pamiers"],
    "Pamiers":["Muret"]}
```

1. Expliquer pourquoi la liste des liaisons suivante n'est pas valide :

```
tableauliaisons = [["Toulouse","Auch"], ["Luchon","Muret"],
["Quillan","Limoux"] ]
```

2. Cette question concerne le joueur n°2 *(Rappel : les liaisons possédées par le joueur n°2 sont représentées par un rectangle blanc dans l'**annexe 2 de l'exercice 1**)*.

a) Donner le tableau `liaisonsJoueur2`, des liaisons possédées par le joueur n°2.

b) Recopier et compléter le dictionnaire suivant, associé au joueur n°2 :

```
DictJoueur2 = {
"Toulouse":["Castres","Castelnaudary"],
…
}
```

3. À partir du tableau de tableaux contenant les liaisons d'un joueur, on souhaite construire le dictionnaire correspondant au joueur. Une première proposition a abouti à la fonction **`construireDict`** ci-dessous.

```
1   def construireDict(listeLiaisons):
2     """
3     listeLiaisons est un tableau de tableaux représentant la
4     liste des liaisons d'un joueur comme décrit dans le problème
5     """
6     Dict={}
7     for liaison in listeLiaisons :
8       villeA = liaison[0]
9       villeB = liaison[1]
10      if not villeA in Dict.keys() :
11        Dict[villeA]=[villeB]
12      else :
13        destinationsA = Dict[villeA]
14        if not villeB in destinationsA :
15          destinationsA.append(villeB)
16    return Dict
```

a) Écrire sur votre copie un assert dans la fonction **`construireDict`** qui permet de vérifier que la **`listeLiaisons`** n'est pas vide.

b) Sur votre copie, donner le résultat de cette fonction ayant comme argument la variable **`liaisonsJoueur1`** donnée dans l'énoncé et expliquer en quoi cette fonction ne répond que partiellement à la demande.

c) La fonction `construireDict`, définie ci-dessus, est donc partiellement inexacte.

Compléter la fonction `construireDict` pour qu'elle génère bien l'ensemble du dictionnaire de tableaux correspondant à la liste de liaisons données en argument. À l'aide des numéros de lignes, on précisera où est inséré ce code.

Exercice 2 (4 points)

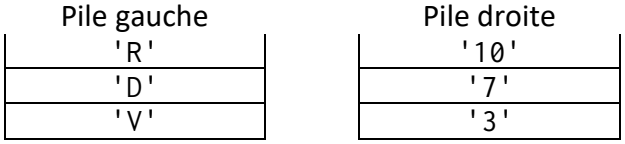
Thème abordé : structures de données : les piles

On cherche à obtenir un mélange d'une liste comportant un nombre **pair** d'éléments. Dans cet exercice, on notera N le nombre d'éléments de la liste à mélanger.

La méthode de mélange utilisée dans cette partie est inspirée d'un mélange de jeux de cartes :

- On sépare la liste en deux piles :
  - ⇨ à gauche, la première pile contient les N/2 premiers éléments de la liste ;
  - ⇨ à droite, la deuxième pile contient les N/2 derniers éléments de la liste.
- On crée une liste vide.
- On prend alors le sommet de la pile de gauche et on le met en début de liste.
- On prend ensuite le sommet de la pile de droite que l'on ajoute à la liste et ainsi de suite jusqu'à ce que les piles soient vides.

Par exemple, si on applique cette méthode de mélange à la liste `['V','D','R','3','7','10']`, on obtient pour le partage de la liste en 2 piles :



La nouvelle liste à la fin du mélange sera donc `['R','10','D','7','V','3']`.

1. Que devient la liste `['7','8','9','10','V','D','R','A']` si on lui applique cette méthode de mélange ?

On considère que l'on dispose de la structure de données de type pile, munie des seules instructions suivantes :

`p = Pile()` : crée une pile vide nommée `p`

`p.est_vide()` : renvoie Vrai si la liste est vide, Faux sinon

`p.empiler(e)` : ajoute l'élément `e` dans la pile

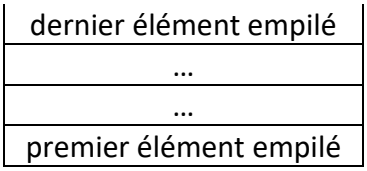
`e = p.depiler()` : retire le dernier élément ajouté dans la pile et le retourne (et l'affecte à la variable `e`)

`p2 = p.copier()` : renvoie une copie de la pile `p` sans modifier la pile `p` et l'affecte à une nouvelle pile `p2`

2. Recopier et compléter le code de la fonction suivante qui transforme une liste en pile.

```
def liste_vers_pile(L):
    '''prend en paramètre une liste et renvoie une
pile'''
    N = len(L)
    p_temp = Pile()
    for i in range(N):
      …….
    return …….
```

3. On considère la fonction suivante qui partage une liste en deux piles. Lors de sa mise au point et pour aider au débuggage, des appels à la fonction `affichage_pile` ont été insérés. La fonction `affichage_pile(p)` affiche la pile `p` à l'écran verticalement sous la forme suivante :




```
def partage(L):
    N = len(L)
    p_gauche = Pile()
    p_droite = Pile()
    for i in range(N/2):
        p_gauche.empile(L[i])
    for i in range(N/2,N):
        p_droite.empile(L[i])
    affichage_pile(p_gauche)
    affichage_pile(p_droite)
    return p_gauche, p_droite
```

Quels affichages obtient-on à l'écran lors de l'exécution de l'instruction : `partage([1,2,3,4,5,6])` ?

4.

4.a Dans un cas général et en vous appuyant sur une séquence de schémas, **expliquer** en quelques lignes comment fusionner deux piles `p_gauche` et `p_droite` pour former une liste `L` en alternant un à un les éléments de la pile `p_gauche` et de la pile `p_droite`.

4.b. **Écrire** une fonction `fusion(p1,p2)` qui renvoie une liste construite à partir des deux piles `p1` et `p2`.

5. Compléter la dernière ligne du code de la fonction `affichage_pile` pour qu'elle fonctionne de manière récursive.

```
def affichage_pile(p):
  p_temp = p.copier()
  if p_temp.est_vide():
    print('____')
  else:
    elt = p_temp.depiler()
    print('| ', elt, ' |'))
    ...                          # ligne à compléter
```

EXERCICE 3 (4 points)

Cet exercice porte sur les arbres binaires de recherche.

Dans cet exercice, les arbres binaires de recherche ne peuvent pas comporter plusieurs fois la même clé. De plus, un arbre binaire de recherche limité à un nœud a une hauteur de 1.
On considère l'arbre binaire de recherche représenté ci-dessous (figure 1), où **val** représente un entier :

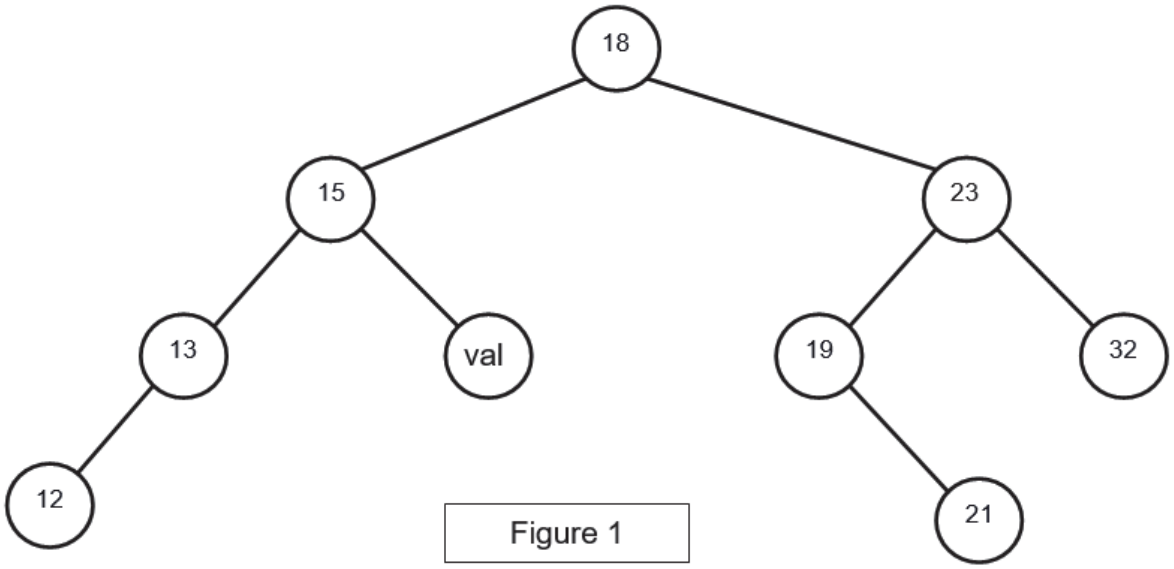


Figure 1

1.

- **a.** Donner le nombre de feuilles de cet arbre et préciser leur valeur (étiquette).
- **b.** Donner le sous arbre-gauche du nœud 23.
- **c.** Donner la hauteur et la taille de l'arbre.
- **d.** Donner les valeurs entières possibles de val pour cet arbre binaire de recherche.

On suppose, pour la suite de cet exercice, que **val** est égal à 16.

2. On rappelle qu'un parcours infixe depuis un nœud consiste, dans l'ordre, à faire un parcours infixe sur le sous arbre-gauche, afficher le nœud puis faire un parcours infixe sur le sous-arbre droit.
Dans le cas d'un parcours suffixe, on fait un parcours suffixe sur le sous-arbre gauche puis un parcours suffixe sur le sous-arbre droit, avant d'afficher le nœud.

- **a.** Donner les valeurs d'affichage des nœuds dans le cas du parcours infixe de l'arbre.
- **b.** Donner les valeurs d'affichage des nœuds dans le cas du parcours suffixe de l'arbre.

3. On considère la classe `Noeud` définie de la façon suivante en Python :

```
class Noeud():
    def __init__(self, v):
        self.ag = None
        self.ad = None
        self.v = v

    def insere(self, v):
        n = self
        est_insere = False
        while not est_insere :
            if v == n.v:
                est_insere = True          } Bloc 1
            elif v < n.v:
                if n.ag != None:           }
                    n = n.ag               }
                else:                      } Bloc 2
                    n.ag = Noeud(v)        }
                    est_insere = True      }
            else:
                if n.ad != None:           }
                    n = n.ad               }
                else:                      } Bloc 3
                    n.ad = Noeud(v)        }
                    est_insere = True      }

    def insere_tout(self, vals):
        for v in vals:
            self.insere(v)
```

a. Représenter l'arbre construit suite à l'exécution de l'instruction suivante :

```
racine = Noeud(18)
racine.insere_tout([12, 13, 15, 16, 19, 21, 32, 23])
```

b. Ecrire les deux instructions permettant de construire l'arbre de la figure 1. On rappelle que le nombre **val** est égal à 16.

c. On considère l'arbre tel qu'il est présenté sur la figure 1. Déterminer l'ordre d'exécution des blocs (repérés de 1 à 3) suite à l'application de la méthode `insere(19)` au nœud racine de cet arbre.

4. Ecrire une méthode `recherche(self, v)` qui prend en argument un entier `v` et renvoie la valeur `True` si cet entier est une étiquette de l'arbre, `False` sinon.

Exercice 4 (4 points)

Gestion d'un club de handball

Thèmes abordés : bases de données

Un club de handball souhaite regrouper efficacement toutes ses informations. Il utilise pour cela des bases de données relationnelles afin d'avoir accès aux informations classiques sur les licenciés du club ainsi que sur les matchs du championnat. Le langage SQL a été retenu.

On suppose dans l'exercice que tous les joueurs d'une équipe jouent à chaque match de l'équipe.

La structure de la base de données est composée des deux tables (ou relations) suivantes:

Table `licencies`	
Attributs	**Types**
`id_licencie`	INT
`prenom`	VARCHAR
`nom`	VARCHAR
`annee_naissance`	INT
`equipe`	VARCHAR

Table `matchs`	
Attributs	**Types**
`id_matchs`	INT
`equipe`	VARCHAR
`adversaire`	VARCHAR
`lieu`	VARCHAR
`date`	DATE

Ci-dessous un exemple de ce que l'on peut trouver dans la base de données :
Exemple **non exhaustif** d'entrées de la table `licencies`

`id_licencie`	`prenom`	`nom`	`annee_naissance`	`equipe`
63	Jean-Pierre	Masclef	1965	Vétérans
102	Eva	Cujon	1992	Femmes 1
125	Emile	Alinio	2000	Hommes 2
247	Ulysse	Trentain	2008	-12 ans

Exemple **non exhaustif** d'entrées de la table `matchs`

`id_match`	`equipe`	`adversaire`	`lieu`	`date`
746	-16 ans	PHC	Domicile	2021-06-19
780	Vétérans	PHC	Exterieur	2021-06-26
936	Hommes 3	LSC	Exterieur	2021-06-20
1032	-19 ans	LOH	Exterieur	2021-05-22
1485	Femmes 2	CHM	Domicile	2021-05-02
1512	Vétérans	ATC	Domicile	2021-04-12

1.

1.a. L'attribut `nom` de la table `licencies` pourrait-il servir de clé primaire ? **Justifier**.

1.b. **Citer** un autre attribut de cette table qui pourrait servir de clé primaire.

2.

2.a. **Expliquer** ce que renvoie la requête SQL suivante :

```
SELECT prenom,nom FROM licencies WHERE equipe ="-12ans"
```

2.b. **Que renvoie** la requête précédente si `prenom,nom` est remplacé par une étoile (*****) ?

2.c. **Ecrire** la requête qui permet l'affichage des dates de tous les matchs joués à domicile de l'équipe *Vétérans*.

3. **Ecrire** la requête qui permet d'inscrire dans la table `licencies`, *Jean Lavenu* né en *2001* de l'équipe *Hommes 2* et qui aura comme numéro de licence *287* dans ce club.

4. On souhaite mettre à jour les données de la table `licencies` du joueur *Joseph Cuviller*, déjà inscrit. Il était en équipe *Hommes 2* et il est maintenant en équipe *Vétérans*. Afin de modifier la table dans ce sens, **proposer** la requête adéquate.

5. Pour obtenir le nom de tous les licenciés qui jouent contre le LSC le 19 juin 2021, **recopier et compléter** la requête suivante :

```
SELECT nom FROM licencies
JOIN Matchs ON licencies.equipe = matchs.equipe
WHERE …… ;
```

Exercice 5 (4 points)

Cet exercice porte sur l'algorithmique et la programmation en Python. Il aborde les notions de tableaux de tableaux et d'algorithmes de parcours de tableaux.

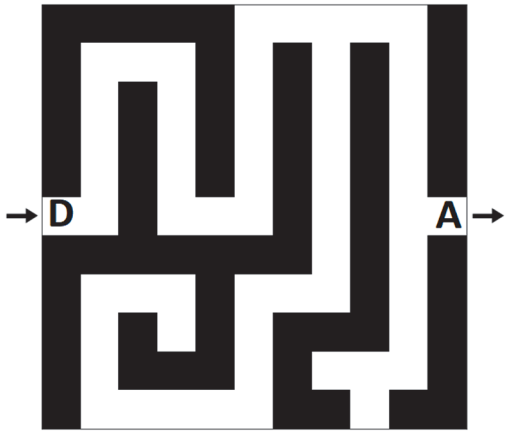
Partie A : Représentation d'un labyrinthe

On modélise un labyrinthe par un tableau à deux dimensions à n lignes et m colonnes avec n et m des entiers strictement positifs.
Les lignes sont numérotées de 0 à $n-1$ et les colonnes de 0 à $m-1$.
La case en haut à gauche est repérée par $(0,0)$ et la case en bas à droite par $(n-1, m-1)$.

Dans ce tableau :

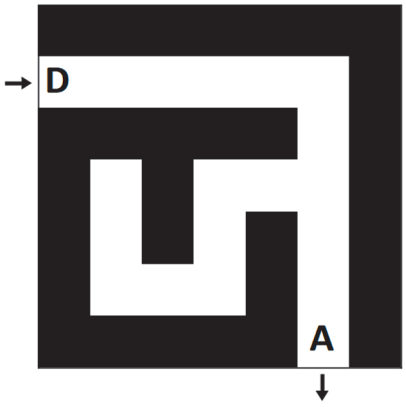
- 0 représente une case vide, hors case de départ et arrivée,
- 1 représente un mur,
- 2 représente le départ du labyrinthe,
- 3 représente l'arrivée du labyrinthe.

Ainsi, en Python, le labyrinthe ci-dessous est représentée par le tableau de tableaux `lab1`.




```
lab1 = [[1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1],
        [1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
        [2, 0, 1, 0, 0, 0, 1, 0, 1, 0, 3],
        [1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1],
        [1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1],
        [1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1],
        [1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1]]
```

1. Le labyrinthe ci-dessous est censé être représenté par le tableau de tableaux `lab2`.
Cependant, dans ce tableau, un mur se trouve à la place du départ du labyrinthe.
Donner une instruction permettant de placer le départ au bon endroit dans `lab2`.




```
lab2 = [[1, 1, 1, 1, 1, 1, 1],
        [1, 0, 0, 0, 0, 0, 1],
        [1, 1, 1, 1, 1, 0, 1],
        [1, 0, 1, 0, 0, 0, 1],
        [1, 0, 1, 0, 1, 0, 1],
        [1, 0, 0, 0, 1, 0, 1],
        [1, 1, 1, 1, 1, 3, 1]]
```

2. Écrire une fonction `est_valide(i, j, n, m)` qui renvoie `True` si le couple (i, j) correspond à des coordonnées valides pour un labyrinthe de taille (n, m), et `False` sinon. On donne ci-dessous des exemples d'appels.

```
>>> est_valide(5, 2, 10, 10)
True
>>> est_valide(-3, 4, 10, 10)
False
```

3. On suppose que le départ d'un labyrinthe est toujours indiqué, mais on ne fait aucune supposition sur son emplacement. Compléter la fonction `depart(lab)` ci-dessous de sorte qu'elle renvoie, sous la forme d'un tuple, les coordonnées du départ d'un labyrinthe (représenté par le paramètre `lab`). Par exemple, l'appel `depart(lab1)` doit renvoyer le tuple `(5, 0)`.

```
def depart(lab) :
    n = len(lab)
    m = len(lab[0])
    …
```

4. Écrire une fonction `nb_cases_vides(lab)` qui renvoie le nombre de cases vides d'un labyrinthe (comprenant donc l'arrivée et le départ).
Par exemple, l'appel `nb_cases_vides(lab2)` doit renvoyer la valeur `19`.

Partie B : Recherche d'une solution dans un labyrinthe

On suppose dans cette partie que les labyrinthes possèdent un unique chemin allant du départ à l'arrivée sans repasser par la même case. Dans la suite, c'est ce chemin que l'on appellera solution du labyrinthe.

Pour déterminer la solution d'un labyrinthe, on parcourt les cases vides de proche en proche. Lors d'un tel parcours, afin d'éviter de tourner en rond, on choisit de marquer les cases visitées. Pour cela, on remplace la valeur d'une case visitée dans le tableau représentant le labyrinthe par la valeur 4.

1. On dit que deux cases d'un labyrinthe sont voisines si elles ont un côté commun.
On considère une fonction `voisines(i, j, lab)` qui prend en arguments deux entiers i et j représentant les coordonnées d'une case et un tableau `lab` qui représente un labyrinthe. Cette fonction renvoie la liste des coordonnées des cases voisines de la case de coordonnées (i, j) qui sont valides, non visitées et qui ne sont pas des murs. L'ordre des éléments de cette liste n'importe pas.

Ainsi, l'appel `voisines(1, 1, [[1, 1, 1], [4, 0, 0], [1, 0, 1]])` renvoie la liste `[(2, 1), (1, 2)]`.

Que renvoie l'appel `voisines(1, 2, [[1, 1, 4], [0, 0, 0], [1, 1, 0]])` ?

2. On souhaite stocker la solution dans une liste `chemin`. Cette liste contiendra les coordonnées des cases de la solution, dans l'ordre. Pour cela, on procède de la façon suivante.

- Initialement :
  - déterminer les coordonnées du départ : c'est la première case à visiter ;
  - ajouter les coordonnées de la case départ à la liste `chemin`.
- Tant que l'arrivée n'a pas été atteinte :
  - on marque la case visitée avec la valeur 4 ;
  - si la case visitée possède une case voisine libre, la première case de la liste renvoyée par la fonction `voisines` devient la prochaine case à visiter et on ajoute à la liste `chemin` ;
  - sinon, il s'agit d'une impasse. On supprime alors la dernière case dans la liste `chemin`. La prochaine case à visiter est celle qui est désormais en dernière position de la liste `chemin`.

a. Le tableau de tableaux `lab3` ci-dessous représente un labyrinthe.

```
lab3 = [[1, 1, 1, 1, 1, 1],
        [2, 0, 0, 0, 0, 3],
        [1, 0, 1, 0, 1, 1],
        [1, 1, 1, 0, 0, 1]]
```

La suite d'instructions ci-dessous simule le début des modifications subies par la liste `chemin` lorsque l'on applique la méthode présentée.

```
# entrée: (1, 0), sortie (1, 5)
chemin = [(1, 0)]
chemin.append((1, 1))
chemin.append((2, 1))
chemin.pop()
chemin.append((1, 2))
chemin.append((1, 3))
chemin.append((2, 3))
```

Compléter cette suite d'instructions jusqu'à ce que la liste `chemin` représente la solution. *Rappel : la méthode* `pop` *supprime le dernier élément d'une liste et renvoie cet élément.*

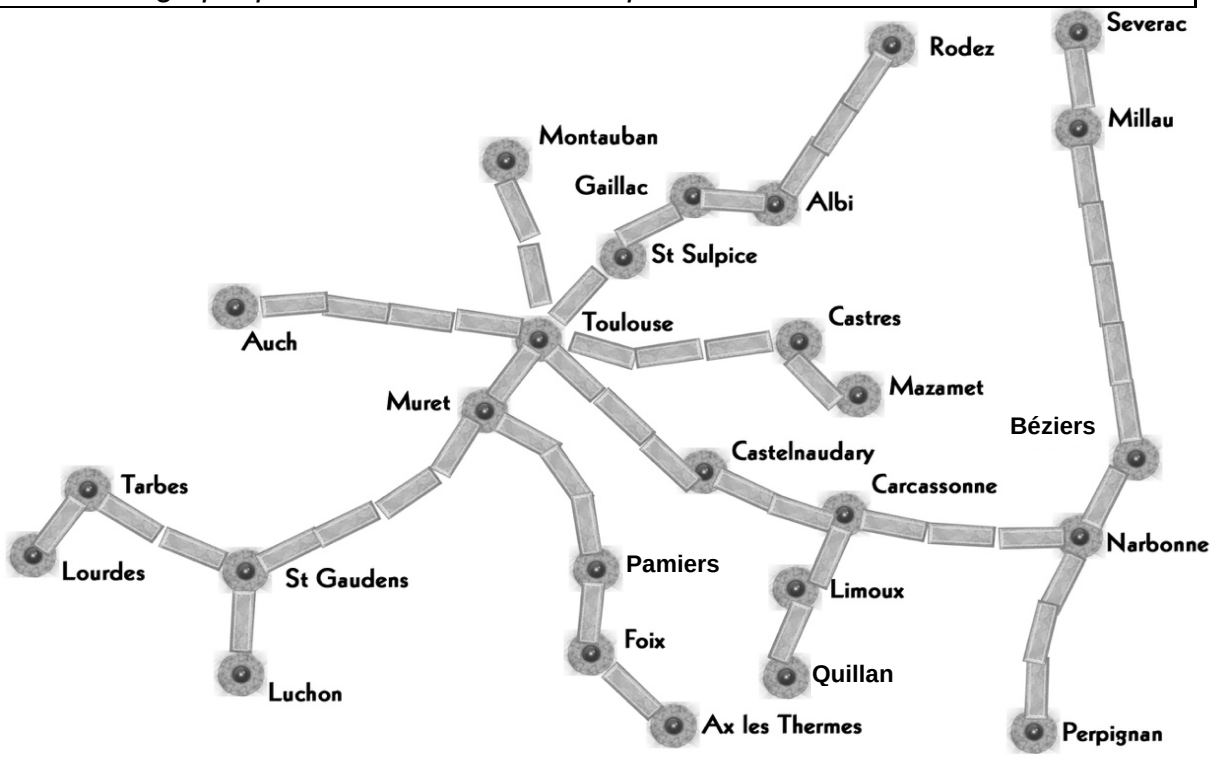
b. Recopier et compléter la fonction `solution(lab)` donnée ci-dessous de sorte qu'elle renvoie le chemin solution du labyrinthe représenté par le paramètre `lab`.
On pourra pour cela utiliser la fonction `voisines`.

```
def solution(lab):
    chemin = [depart(lab)]
    case = chemin[0]
    i = case[0]
    j = case[1]
    ...
```

Par exemple, l 'appel `solution(lab2)` doit renvoyer `[(1, 0), (1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (2, 5), (3, 5), (4, 5), (5, 5), (6, 5)]`.

Exercice 1 - Annexe 1

graphique 1 : Extrait des liaisons possibles en Occitanie



Exercice 1 - Annexe 2

Liaisons possédées par le joueur 1 (en noir) et le joueur 2 (en blanc)