Métropole - mai 2022 - sujet 2

Exercice 1 (ABR, POO et récursivité - 4 points)
Dans cet exercice, la taille d'un arbre est le nombre de nœuds qu'il contient. Sa hauteur est le nombre de nœuds du plus long chemin qui joint le nœud racine à l'une des feuilles (nœuds sans sous-arbres). On convient que la hauteur d'un arbre ne contenant qu'un nœud vaut 1 et la hauteur de l'arbre vide vaut 0.

1. On considère l'arbre binaire représenté ci-dessous :

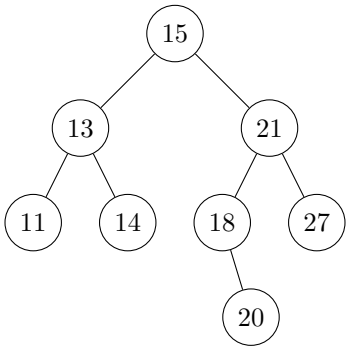


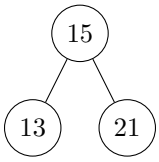
FIGURE 1

   (a) Donner la taille de cet arbre.
   (b) Donner la hauteur de cet arbre.
   (c) Représenter sur la copie le sous-arbre droit du nœud de valeur 15.
   (d) Justifier que l'arbre de la figure 1 est un arbre binaire de recherche.
   (e) On insère la valeur 17 dans l'arbre de la figure 1 de telle sorte que 17 soit une nouvelle feuille de l'arbre et que le nouvel arbre obtenu soit encore un arbre binaire de recherche. Représenter sur la copie ce nouvel arbre.

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

```
1 class Noeud:
2     def __init__(self, g, v, d):
3         self.gauche = g
4         self.valeur = v
5         self.droit = d
```

   (a) Parmi les trois instructions **(A)**, **(B)** et **(C)** suivantes, écrire sur la copie la lettre correspondant à celle qui construit et stocke dans la variable `abr` l'arbre représenté ci-dessous.



   (A) `abr=Noeud(Noeud(Noeud(None,13,None),15,None),21,None)`
   (B) `abr=Noeud(None,13,Noeud(Noeud(None,15,None),21,None))`
   (C) `abr=Noeud(Noeud(None,13,None),15,Noeud(None,21,None))`

   (b) Recopier et compléter la ligne 7 du code de la fonction `ins` ci-dessous qui prend en paramètres une valeur `v` et un arbre binaire de recherche `abr` et qui renvoie l'arbre obtenu suite à l'insertion de la valeur `v` dans l'arbre `abr`. Les lignes 8 et 9 permettent de ne pas insérer la valeur `v` si celle-ci est déjà présente dans `abr`.

```
def ins(v, abr):
    if abr is None:
        return Noeud(None, v, None)
    if v > abr.valeur:
        return Noeud(abr.gauche,abr.valeur,ins(v,abr.droit))
    elif v < abr.valeur:
        return ........................................
    else:
        return abr
```

3. La fonction `nb_sup` prend en paramètres une valeur `v` et un arbre binaire de recherche `abr` et renvoie le nombre de valeurs supérieures ou égales à la valeur `v` dans l'arbre `abr`.
Le code de cette fonction `nb_sup` est donné ci-dessous :

```
def nb_sup(v, abr):
    if abr is None:
        return 0
    else:
        if abr.valeur >= v:
            return 1+nb_sup(v, abr.gauche)+nb_sup(v, abr.droit)
        else:
            return nb_sup(v, abr.gauche)+nb_sup(v, abr.droit)
```

(a) On exécute l'instruction `nb_sup(16, abr)` dans laquelle `abr` est l'arbre initial de la figure 1. Déterminer le nombre d'appels à la fonction `nb_sup`.

(b) L'arbre passé en paramètre étant un arbre binaire de recherche, on peut améliorer la fonction `nb_sup` précédente afin de réduire ce nombre d'appels. Ecrire sur la copie le code modifié de cette fonction.

Exercice 2 (Pile - 4 points)
La poussette est un jeu de cartes en solitaire. Cet exercice propose une version simplifiée de ce jeu basée sur des nombres.

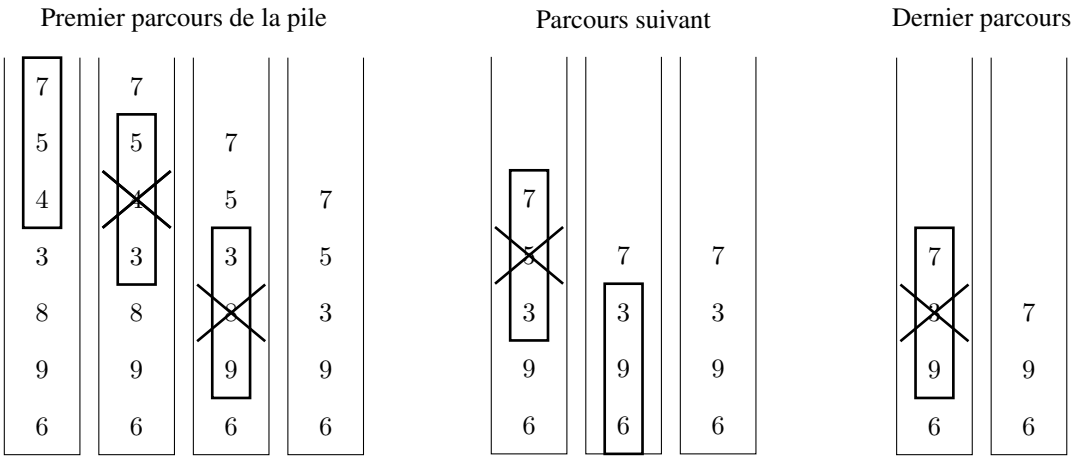
On considère une pile constituée de nombres entiers tirés aléatoirement. Le jeu consiste à réduire la pile suivant la règle suivante : quand la pile contient du haut vers le bas un triplet dont les termes du haut et du bas sont de même parité, on supprime l'élément central.

Par exemple :

- Si la pile contient du haut vers le bas, le triplet $1\ 0\ 3$, on supprime le 0.
- Si la pile contient du haut vers le bas, le triplet $1\ 0\ 8$, la pile reste inchangée.

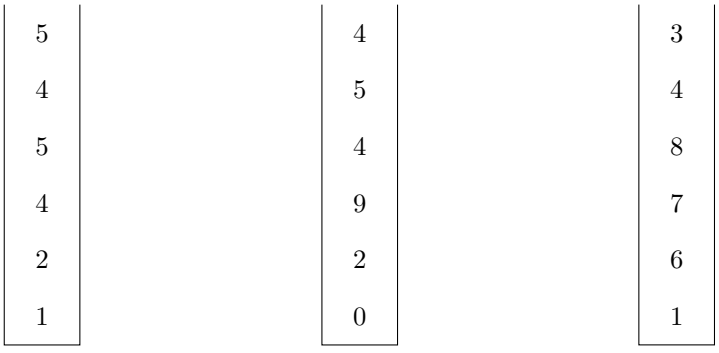
On parcourt la pile ainsi de haut en bas et on procède aux réductions. Arrivé en bas de la pile, on recommence la réduction en repartant du sommet de la pile jusqu'à ce que la pile ne soit plus réductible. Une partie est « gagnante » lorsque la pile finale est réduite à deux éléments exactement.

Voici un exemple détaillé de déroulement d'une partie.



1. (a) Donner les différentes étapes de réduction de la pile suivante :

(b) Parmi les piles proposées ci-dessous, donner celle qui est gagnante.



Pile A Pile B Pile C

L'interface d'une pile est proposée ci-dessous. On utilisera uniquement les fonctions figurant dans le tableau suivant :

Structure de données abstraite : Pile

- `creer_pile_vide()` renvoie une pile vide
- `est_vide(p)` renvoie `True` si `p` est vide, `False` sinon
- `empiler(p, element)` ajoute `element` au sommet de `p`
- `depiler(p)` retire l'élément au sommet de `p` et le renvoie
- `sommet(p)` renvoie l'élément au sommet de `p` sans le retirer de `p`
- `taille(p)` renvoie le nombre d'éléments de `p`

2. La fonction `reduire_triplet_au_sommet` permet de supprimer l'élément central des trois premiers éléments en partant du haut de la pile, si l'élément du bas et du haut sont de même parité. Les éléments dépilés et non supprimés sont replacés dans le bon ordre dans la pile.
Recopier et compléter sur la copie le code de la fonction `reduire_triplet_au_sommet` prenant une pile `p` en paramètre et qui la modifie en place. Cette fonction ne renvoie donc rien.

```
1  def reduire_triplet_au_sommet(p):
2      a = depiler(p)
3      b = depiler(p)
4      c = sommet(p)
5      if a % 2 != .... :
6          empiler(p, ...)
7      empiler(p, ...)
```

3. On se propose maintenant d'écrire une fonction `parcourir_pile_en_reduisant` qui parcourt la pile du haut vers le bas en procédant aux réductions pour chaque triplet rencontré quand cela est possible.
   (a) Donner la taille minimale que doit avoir une pile pour être réductible.
   (b) Recopier et compléter le code suivant sur la copie :

```
1  def parcourir_pile_en_reduisant(p):
2      q = creer_pile_vide()
3      while taille(p) >= ....:
4          reduire_triplet_au_sommet(p)
5          e = depiler(p)
6          empiler(q, e)
7      while not est_vide(q):
8          ............
9          ............
10     return p
```

4. Partant d'une pile d'entiers `p`, on propose ici d'implémenter une fonction récursive `jouer` renvoyant la pile `p` entièrement simplifiée. Une fois la pile parcourue de haut en bas et réduite, on procède à nouveau à sa réduction à condition que cela soit possible. Ainsi :
   - si la pile `p` n'a pas subi de réduction, on la renvoie ;
   - sinon on appelle à nouveau la fonction `jouer`, prenant en paramètre la pile réduite.

   Recopier et compléter sur la copie le code ci-dessous :

```
1  def jouer(p):
2      q = parcourir_pile_en_reduisant(p)
3      if ............. :
4          return p
5      else:
6          return jouer(...)
```

Exercice 3 (Réseaux et protocoles de routage - 4 points)
Rappels :

- Une adresse IPv4 est composée de 4 octets, soit 32 bits. Elle est notée `a.b.c.d`, où `a`, `b`, `c` et `d` sont les valeurs des 4 octets.
- La notation `a.b.c.d/n` signifie que les `n` premiers bits de l'adresse IP représentent la partie « réseau », les bits qui suivent représentent la partie « machine ».
- L'adresse IPv4 dont tous les bits de la partie « machine » sont à `0` est appelée « adresse du réseau ».
- L'adresse IPv4 dont tous les bits de la partie « machine » sont à `1` est appelée « adresse de diffusion ».

On considère le réseau représenté sur la Figure 1 ci-dessous :

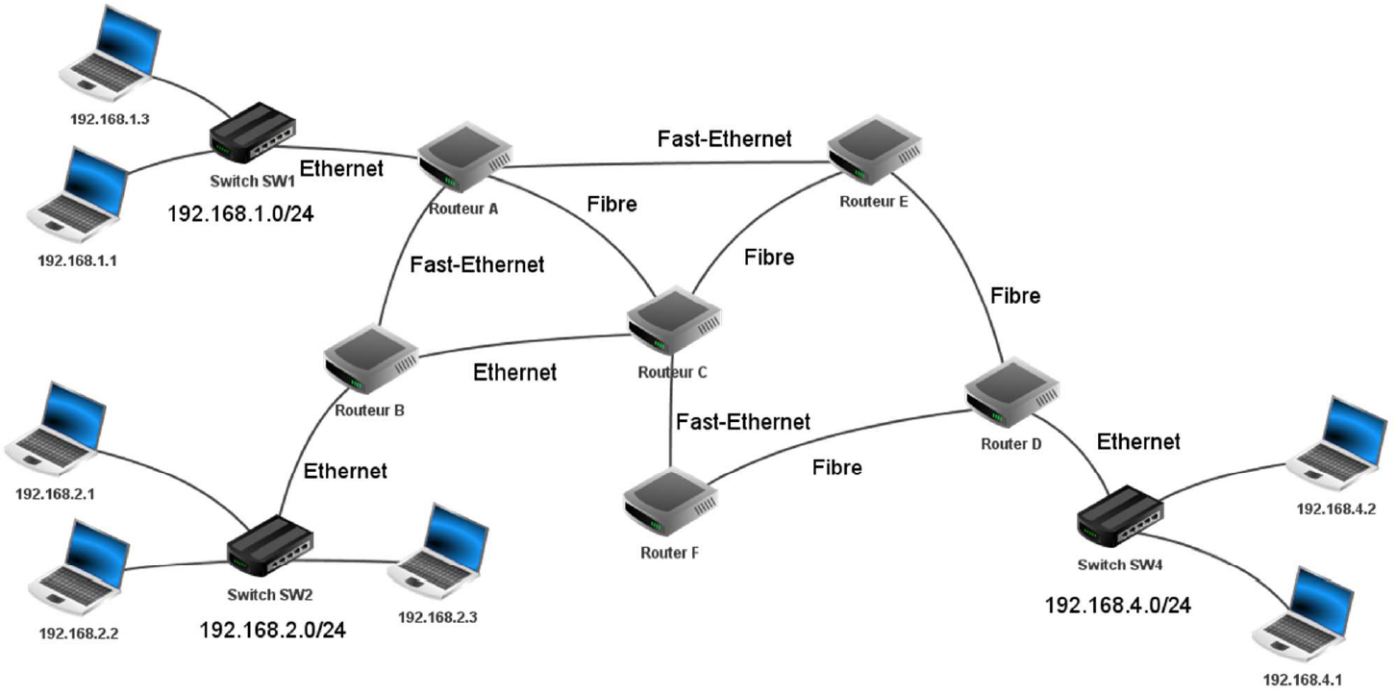


FIGURE 1 – schéma d'un réseau

1. On considère la machine d'adresse IPv4 `192.168.1.1/24`
   (a) Donner l'adresse du réseau sur lequel se trouve cette machine.
   (b) Donner l'adresse de diffusion (broadcast) de ce réseau.
   (c) Donner le nombre maximal de machines que l'on peut connecter sur ce réseau.
   (d) On souhaite ajouter une machine sur ce réseau, proposer une adresse IPv4 possible pour cette machine.
2. (a) La machine d'adresse IPv4 `192.168.1.1` transmet un paquet IPv4 à la machine d'adresse IPv4 `192.168.4.2` Donner toutes les routes pouvant être empruntées par ce paquet IPv4, chaque routeur ne pouvant être traversé qu'une seule fois.
   (b) Expliquer l'utilité d'avoir plusieurs routes possibles reliant les réseaux `192.168.1.0/24` et `192.168.4.0/24`
3. Dans cette question, on suppose que le protocole de routage mis en place dans le réseau est RIP. Ce protocole consiste à minimiser le nombre de sauts. Le schéma du réseau est celui de la figure 1. Les tables de routage utilisées sont données ci-dessous :

Routeur A	
Destination	passe par
B	...
C	...
D	E
E	...
F	C

Routeur B	
Destination	passe par
A	A
C	C
D	C
E	C
F	C

Routeur C	
Destination	passe par
A	A
B	B
D	E
E	E
F	F

Routeur D	
Destination	passe par
A	E
B	F
C	F
E	E
F	F

Routeur E	
Destination	passe par
A	A
B	C
C	C
D	D
F	C

Routeur F	
Destination	passe par
A	C
B	C
C	C
D	D
E	C

Tables de routage

(a) Recopier et compléter sur la copie la table de routage du routeur A.

(b) Un paquet IP doit aller du routeur B au routeur D. En utilisant les tables de routage, donner le parcours emprunté par celui-ci.

(c) Les connexions entre les routeurs B-C et A-E étant coupées, sur la copie, réécrire les tables de routage des routeurs A, B et C.

(d) Déterminer le nouveau parcours emprunté par le paquet IP pour aller du routeur B au routeur D.

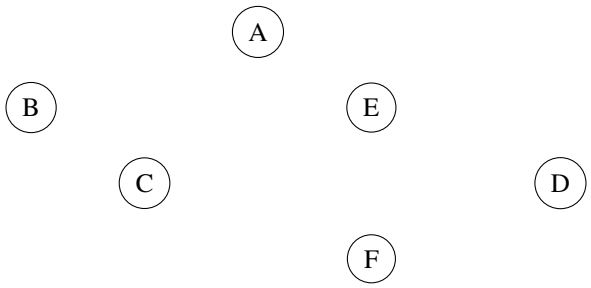
4. Dans cette question, on suppose que le protocole de routage mis en place dans le réseau est OSPF. Ce protocole consiste à minimiser la somme des coûts des liaisons empruntées. Le coût d'une liaison est défini par la relation

$$\text{coût } = \frac{10^8}{d},$$

où d représente le débit en bit/s et coût est sans unité. Le schéma du réseau est celui de la figure 1.

(a) Déterminer le coût des liaisons Ethernet ($d = 10^7$ bit/s), Fast-Ethernet ($d = 10^8$ bit/s) et Fibre ($d = 10^9$ bit/s).

(b) On veut représenter schématiquement le réseau de routeurs à partir du schéma du réseau figure 1.
Recopier sur la copie le schéma ci-dessous et tracer les liaisons entre les routeurs en y indiquant le coût.



(c) Un paquet IPv4 doit être acheminé d'une machine ayant pour adresse IPv4 `192.168.2.1` à une machine ayant pour adresse IPv4 `192.168.4.1`. Ecrire les routes possibles, c'est-à-dire la liste des routeurs traversés, et le coût de chacune de ces routes, chaque routeur ne pouvant être traversé qu'une seule fois.

(d) Donner, en la justifiant, la route qui sera empruntée par un paquet IPv4 pour aller d'une machine ayant pour adresse IPv4 `192.168.2.1` à une machine ayant pour adresse IPv4 `192.168.4.1`

Exercice 4 (Bases de données et SQL - 4 points)

L'énoncé de cet exercice utilise les mots clefs du langage SQL suivants : `SELECT`, `FROM`, `WHERE`, `JOIN ON`, `UPDATE`, `SET`, `INSERT INTO VALUES`, `COUNT`, `ORDER BY`.

- La clause `ORDER BY` suivie d'un attribut permet de trier les résultats par ordre croissant de l'attribut ;
- `COUNT(*)` renvoie le nombre de lignes d'une requête.

Un musicien souhaite créer une base de données relationnelle contenant ses morceaux et interprètes préférés. Pour cela, il utilise le langage SQL. Il crée une table `morceaux` qui contient entre autres les titres des morceaux et leur année de sortie :

id_morceau	titre	annee	id_interprete
1	Like a Rolling Stone	1965	1
2	Respect	1967	2
3	Imagine	1970	3
4	Hey Jude	1968	4
5	Smells Like Teen Spirit	1991	5
6	I Want To hold Your Hand	1963	4

Il crée la table `interpretes` qui contient les interprètes et leur pays d'origine :

id_interprete	nom	pays
1	Bob Dylan	États-Unis
2	Aretha Franklin	États-Unis
3	John Lennon	Angleterre
4	The Beatles	Angleterre
5	Nirvana	États-Unis

`id_morceau` de la table `morceaux` et `id_interprete` de la table `interpretes` sont des clés primaires.
L'attribut `id_interprete` de la table `morceaux` fait directement référence à la clé primaire de la table `interpretes`.

1. (a) Ecrire le résultat de la requête suivante :

```
SELECT titre FROM morceaux WHERE id_interprete = 4;
```

   (b) Ecrire une requête permettant d'afficher les noms des interprètes originaires d'Angleterre.

   (c) Ecrire le résultat de la requête suivante :

```
SELECT titre, annee FROM morceaux ORDER BY annee;
```

   (d) Ecrire une requête permettant de calculer le nombre de morceaux dans la table `morceaux`.

   (e) Ecrire une requête affichant les titres des morceaux par ordre alphabétique.

2. (a) Citer, en justifiant, la clé étrangère de la table `morceaux`.

   (b) Ecrire un schéma relationnel des tables `interpretes` et `morceaux`.

   (c) Expliquer pourquoi la requête suivante produit une erreur :

```
INSERT INTO interpretes VALUES (1, 'Trust', 'France');
```

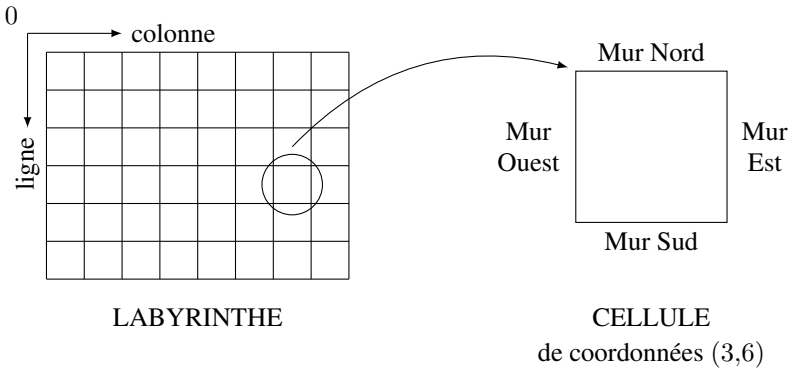
3. (a) Une erreur de saisie a été faite. Ecrire une requête SQL permettant de changer l'année du titre `Imagine` en 1971.

   (b) Ecrire une requête SQL permettant d'ajouter l'interprète `The Who` venant d'Angleterre à la table `interpretes`. On lui donnera un `id_interprete` égal à `6`.

   (c) Ecrire une requête SQL permettant d'ajouter le titre `My Generation` de `The Who` à la table `morceaux`. Ce titre est sorti en `1965` et on lui donnera un `id_morceau` de `7` ainsi que l' `id_interprete` qui conviendra.

4. Ecrire une requête permettant de lister les titres des interprètes venant des États-Unis.

Exercice 5 (POO et méthode « diviser pour régner » - 4 points)



Un labyrinthe est composé de cellules possédant chacune quatre murs (voir cidessus). La cellule en haut à gauche du labyrinthe est de coordonnées $(0,0)$. On définit la classe `Cellule` ci-dessous. Le constructeur possède un attribut `murs` de type `dict` dont les clés sont `'N'`, `'E'`, `'S'` et `'O'` et dont les valeurs sont des booléens (`True` si le mur est présent et `False` sinon).

```
class Cellule:
    def __init__(self, murNord, murEst, murSud, murOuest):
        self.murs={'N':murNord,'E':murEst,'S':murSud,'O':murOuest}
```

1. Recopier et compléter sur la copie l'instruction Python suivante permettant de créer une instance `cellule` de la classe `Cellule` possédant tous ses murs sauf le mur Est.

```
cellule = Cellule(...)
```

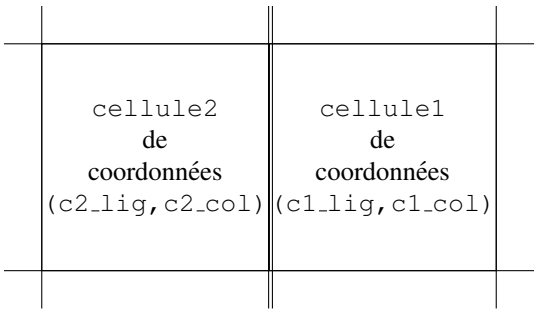
2. Le constructeur de la classe `Labyrinthe` ci-dessous possède un seul attribut `grille`. La méthode `construire_grille` permet de construire un tableau à deux dimensions `hauteur` et `longueur` contenant des cellules possédant chacune ses quatre murs. Recopier et compléter sur la copie les lignes 6 à 10 de la classe `Labyrinthe`.

```
1  class Labyrinthe:
2      def __init__(self, hauteur, longueur):
3          self.grille=self.construire_grille(hauteur, longueur)
4      def construire_grille(self, hauteur, longueur):
5          grille = []
6          for i in range(...):
7              ligne = []
8              for j in range(...):
9                  cellule = ...
10                 ligne.append(...)
11             grille.append(ligne)
12         return grille
```

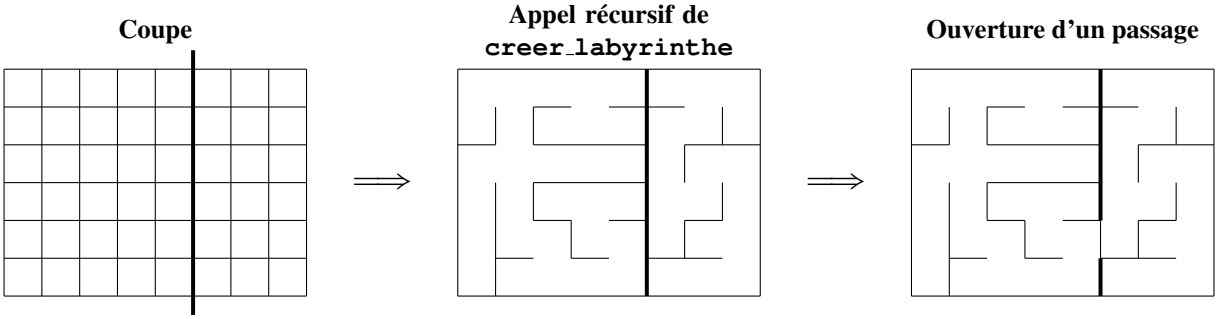
Pour générer un labyrinthe, on munit la classe `Labyrinthe` d'une méthode `creer_passage` permettant de supprimer des murs entre deux cellules ayant un côté commun afin de créer un passage. Cette méthode prend en paramètres les coordonnées `(c1_lig, c1_col)` d'une cellule notée `cellule1` et les coordonnées `(c2_lig, c2_col)` d'une cellule notée `cellule2` et crée un passage entre `cellule1` et `cellule2`.

```
1  def creer_passage(self, c1_lig, c1_col, c2_lig, c2_col):
2      cellule1 = self.grille[c1_lig][c1_col]
3      cellule2 = self.grille[c2_lig][c2_col]
4      # cellule2 au Nord de cellule1
5      if c1_lig - c2_lig == 1 and c1_col == c2_col:
6          cellule1.murs['N'] = False
7          ....
8      # cellule2 à lOuest de cellule1
9      elif ....
10         ....
11         ....
```

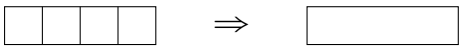
3. La ligne 6 permet de supprimer le mur Nord de `cellule1`. Un mur de `cellule2` doit aussi être supprimé pour libérer un passage entre `cellule1` et `cellule2`. Ecrire l'instruction Python que l'on doit ajouter à la ligne 7.
4. Recopier et compléter sur la copie le code Python des lignes 9 à 11 qui permettent le traitement du cas où `cellule2` est à l'Ouest de `cellule1` :



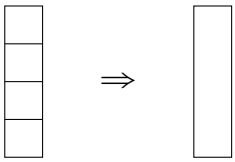
Pour créer un labyrinthe, on utilise la méthode « diviser pour régner » en appliquant récursivement l'algorithme `creer_labyrinthe` sur des sous-grilles obtenues en coupant la grille en deux puis en reliant les deux sous-labyrinthes en créant un passage entre eux.



La méthode `creer_labyrinthe` permet, à partir d'une grille, de créer un labyrinthe de hauteur `haut` et de longueur `long` dont la cellule en haut à gauche est de coordonnées `(ligne, colonne)`.
Le cas de base correspond à la situation où la grille est de hauteur 1 ou de largeur 1.
Il suffit alors de supprimer tous les murs intérieurs de la grille.



Exemple avec
`haut = 1` et `long = 4`



Exemple avec
`haut = 4` et `long = 1`

5. Recopier et compléter sur la copie les lignes 2 à 7 de la méthode `creer_labyrinthe` traitant le cas de base.

```
1 def creer_labyrinthe(self, ligne, colonne, haut, long):
2     if haut == 1 : # Cas de base
3         for k in range(...):
4             self.creer_passage(ligne, k, ligne, k+1)
5     elif long == 1: # Cas de base
6         for k in range(...):
7             self.creer_passage(...)
8     else: # Appels récursifs
9         # Code non étudié (Ne pas compléter)
```

6. Dans cette question, on considère une grille de hauteur `haut = 4` et de longueur `long = 8` dont chaque cellule possède tous ses murs.
   On fixe les deux contraintes supplémentaires suivantes sur la méthode `creer_labyrinthe` :
   - Si `haut` $\geqslant$ `long`, on coupe horizontalement la grille en deux sous-labyrinthes de même dimension.
   - Si `haut` $<$ `long`, on coupe verticalement la grille en deux sous-labyrinthes de même dimension.

L'ouverture du passage entre les deux sous-labyrinthes se fait le plus au Nord pour une coupe verticale et le plus à l'Ouest pour une coupe horizontale.

Dessiner le labyrinthe obtenu suite à l'exécution complète de l'algorithme `creer_labyrinthe` sur cette grille.