

Info2A-PSI - Informatique de Tronc Commun – DS01

Exercice

On considère le jeu de Juniper Green dont les règles sont les suivantes, un entier $N \geq 1$ ayant fixé.

- Règle 1 : Le premier joueur choisi un nombre entre 1 et N .
- Règle 2 : Une fois qu'un nombre a été choisi il ne pourra plus jamais être utilisé.
- Règle 3 : Chaque joueur doit choisir un nombre qui est, soit un diviseur, soit un multiple du nombre précédent.
- Règle 4 : Le joueur qui ne peut plus jouer perd la partie.

Un déroulement possible du jeu, pour $N = 20$ est :

Tour de jeu	Joueur	Nombre choisi
1	1	3
2	2	6
3	1	2
4	2	1
5	1	19

Le joueur 2 a perdu.

On peut représenter le jeu par un graphe non orienté dont les sommets sont étiquetés par les entiers de 0 à N et dans lequel une arête existe entre deux sommets étiquetés a et b si et seulement si a est un multiple ou un diviseur de b .

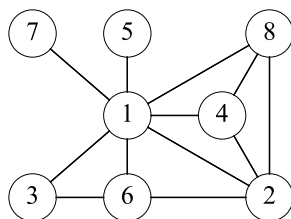
Dans la suite de l'exercice, on écrira « sommet a » pour dire « sommet étiqueté a ».

On notera G_N le graphe non orienté d'ordre N représentant le jeu pour une valeur donnée de N .

On notera M_N la matrice d'adjacence représentant G_N . M_N sera d'ordre N , et, pour tout $1 \leq i, j \leq N$, on aura $M_{i,j} = 1$ s'il existe une arête entre deux sommets a et b et zéro sinon.

La représentation en machine de M_N sera une liste de listes Python, notée ici `M`, et l'existence d'une arête entre deux sommets i et j ($1 \leq i, j \leq N$), sera donnée par la valeur `M[i - 1][j - 1]`.

Ainsi, pour $N = 8$, le jeu sera représenté par le graphe suivant, G_8 :



Q1. Quelle est la taille du graphe G_8 ?

Q2. Dessiner sur votre copie le graphe représentant le jeu pour $N = 4$ et pour $N = 12$.

Q3. Écrire sur votre copie la matrice d'adjacence, M_8 , du graphe G_8 , sous la forme d'une liste de listes Python, que l'on notera `M8`.

Q4. Définir une fonction, `consM`, prenant en argument un entier $N \geq 1$ et renvoyant la représentation de la matrice d'adjacence du graphe G_N sous la forme d'une liste de liste.

Q5. Écrire sur votre copie, un dictionnaire Python, `DA8`, dont les clés sont de type entier et désignent les sommets du graphe G_8 et dont les valeurs sont les listes d'adjacence de chaque sommet, chaque sommet voisin étant représenté par l'entier qui est son étiquette.

On aurait par exemple, pour représenter G_4 , écrit le dictionnaire :

```
DA4 = {1: [2, 3, 4], 2: [1, 4], 3: [1], 4: [1, 2]}
```

Q6. Définir une fonction, `consDA`, prenant en argument un entier $N \geq 1$ et renvoyant la représentation de la matrice d'adjacence du graphe g_N sous la forme d'un dictionnaire des listes d'adjacence de ses sommets. Les listes d'adjacence seront des listes croissantes d'entiers (les numéros des sommets voisins).

La fonction devra faire appel à la fonction `consM` définie en Q4.

Q7. Compléter les deux tableaux en annexe, afin de retracer les sommets visités dans un parcours en largeur ou en profondeur du graphe G_8 , à **partir du sommet 4**.

Lors de ces parcours, les listes d'adjacence devront obligatoirement être explorées selon les numéros de sommets croissants (« de gauche à droite » donc, cf. Q6.).

Q8. Lequel de ces deux parcours est le plus à même de retracer le déroulement d'une partie dans laquelle le joueur 1 aurait joué le nombre 4 à son premier coup.

Préciser dans quelle mesure et pourquoi.

On indiquera où dans le tableau lire la succession des coups joués et par quel joueur, ainsi que la fin de partie, si cela est possible.

On demande d'écrire deux fonctions implémentant, l'un un parcours en largeur, l'autre un parcours en profondeur d'un graphe représenté par un dictionnaire de ses listes d'adjacence.

Une pile sera implémentée par une liste Python, en utilisant les méthodes `.append` et `.pop`.

Pour implémenter des files, on importera le type d'objets `deque` depuis le module `collections`, à l'aide de l'instruction suivante :

```
from collections import deque
```

Une file vide `fd` sera créée par l'instruction

```
fd = deque()
```

Une valeur `x` pourra être enfilée dans une file `fd` à l'aide de l'instruction

```
fd.appendleft(x)
```

et défilée à l'aide de l'instruction

```
fd.pop()
```

On testera si une file `fd` est vide avec le test `len(f)==0`.

Q9. Définir une fonction `DFS` (pour « *Depth First Search* ») implémentant un parcours en profondeur, prenant en argument un dictionnaire, `DA`, représentant un graphe, dont les clés sont les numéros de sommets et les valeurs les listes d'adjacences de ces sommets, et un sommet, s_0 , pris comme origine du parcours, et renvoyant la liste des sommets du graphe dans l'ordre dans lequel on a exploré leurs voisinage (le premier sommet dans la liste sera le sommet s_0 , le dernier, celui dont l'exploration du voisinage a terminé l'algorithme).

Q10. Définir de même une fonction `BFS` (pour « *Breadth First Search* ») pour le parcours en largeur.

PROBLÈME

Consignes Python

Tout code doit être écrit dans le langage Python.

- Veuillez indenter votre code correctement.
- Tout ce dont vous avez besoin (fonctions, méthodes) est indiqué ci-dessous.
- Vous pouvez écrire vos propres fonctions, dans ce cas elles doivent être documentées (on doit savoir ce qu'elles font).

Dans tous les cas, la dernière fonction écrite doit être celle qui répond à la question.

Fonctions et méthodes autorisées

- Itérateurs fournis par la fonction `range`.
- Slicing (extraction de portions) sur les listes.
- La fonction `len` donne la taille d'une liste.
- Les méthodes utilisables sur les listes sont `append` et `pop`, tout autre méthode (`insert`, `sort` ...) est exclue.
- Module et fonctions pour les représentations graphiques :
 - on utilisera le sous-module `pyplot` de la bibliothèque `matplotlib`, que l'on importera par l'instruction `import matplotlib.pyplot as plt`;
 - l'instruction `plt.plot([x0, x1, ..., xn], [y0, y1, ..., yn], 'k')` trace la ligne brisée joignant les points d'abscisses respectives $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$, en trait plein noir ('k');
 - si la ligne brisée forme un contour fermé (un polygone), et si `c` est une chaîne de caractère définissant une couleur, on peut tracer ce polygone et colorier son intérieur avec la couleur désignée par une chaîne de caractères `c` avec l'instruction suivante :


```
plt.fill([x0, x1, ..., xn], [y0, y1, ..., yn], c),
```

 (la figure recouvrera les éventuels tracés précédents lorsqu'elle s'y superpose);
 - `plt.axis('equal')` impose que la figure soit tracée dans un repère orthonormé;
 - `plt.show()` déclenche l'ouverture de la fenêtre et le tracé de la figure construite.
- Fonctions pour les fichiers :
 - `f = open(fichier, mode)` ouvre un « wrapper » vers un fichier. Le mode peut être 'r' (lecture), 'w' (écriture) ou 'a' (ajout).
 - `f.close()` ferme le « wrapper ».
 - `f.readlines()` renvoie une liste de chaînes de caractères dont chacune correspond aux caractères composant une ligne du fichier, caractère de saut de ligne inclus;
 - `f.write(s)` : avec `f` un tel « wrapper » vers un fichier ouvert en écriture ou ajout, écrit dans le fichier la chaîne de caractères `s`.
 - `+` : concaténation de chaînes de caractères.
 - `'\n'` : caractère spécial de saut de ligne.

Jeu de la vie sur un univers fini

Le jeu de la vie est un automate cellulaire imaginé par John Horton Conway en 1970. Malgré sa définition *via* des règles très simples, il recèle une incroyable complexité. Ce sujet en présente une variante, où l'univers est fini.

L'univers considéré est un quadrillage de dimension $N \times N$, mais les cases de la première ligne sont voisines de celles de la dernière, de même que celles de la colonne de droite sont voisines de celles de la colonne de gauche. Ainsi, chaque case de l'univers possède exactement 8 voisines (voir figure), adjacentes horizontalement, verticalement, ou diagonalement.

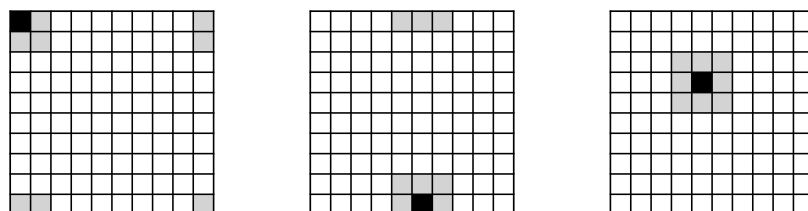


FIGURE 1 : Voisins dans le jeu de la vie sur une grille de taille 10 : en noir une case donnée, en gris ses 8 voisines.

- Q1.** Écrire une séquence d'instructions permettant de tracer et d'afficher un carré de côté 10 représentant le plateau, carré, du jeu de la vie, dans un repère orthonormé.
Le coin inférieur droit du plateau sera pris à l'origine du repère.
On effectuera les importations nécessaires.
- Q2.** Écrire une séquence d'instructions par laquelle on pourrait compléter le script de la question Q1. afin de tracer le quadrillage matérialisant les cent cellules du plateau.
- Q3.** Écrire une séquence d'instructions par laquelle on pourrait compléter le script de la question Q1. pour faire apparaître la cellule noire de la figure à droite ci-dessus, ainsi que les cellules grises qui la bordent.
La couleur grise pourra être désignée par la chaîne de caractères 'lightgray', la couleur noire par l'abrégié 'k'.

Chaque case du jeu de la vie est soit vivante, soit morte. À partir d'une situation initiale au temps t situation du jeu évolue entre les instants t et $t + 1$ en suivant les deux règles suivantes :

- une cellule morte possédant exactement trois cellules voisines vivantes devient vivante (elle naît) ;
- une cellule vivante possédant deux ou trois cellules voisines vivantes le reste, sinon elle meurt.

Représentation en Python. Un univers de taille $N \times N$ est représenté par une liste `U` comportant N listes toutes de taille N . La i -ème liste de `U` (c'est-à-dire `U[i]`, pour $0 \leq i \leq N - 1$) représente la i -ème ligne de la grille en partant du haut, en faisant démarrer l'indexation à zéro. Les colonnes sont également indexées de 0 à $N - 1$. La case indicée par (i, j) (c'est-à-dire `U[i][j]`) contient un booléen indiquant la présence d'une cellule vivante. La figure suivante montre un univers de taille 4×4 et sa représentation sous forme de liste de listes.

```
[[ True, False, False, False],
 [False, False,  True,  True],
 [False, False, False, False],
 [ True, False,  True, False]]
```

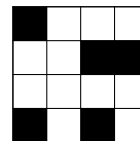


FIGURE 2 : Représentation d'un univers par une liste de listes. Les cellules vivantes sont en noir.

Dans toute la suite, on désigne par univers une telle liste de listes `U` contenant des booléens, qui sera supposée carrée : avec N la taille de `U`, toutes les listes de `U` ont elles-mêmes une taille N . L'entier N sera supposé toujours au moins égal à 3.

On souhaite pouvoir stocker dans un fichier une configuration que l'on trouverait intéressante.

- Q4.** Écrire une fonction `imprime(U, nom)`, prenant en entrée un univers `U` est une chaîne de caractères `nom` (par exemple 'planeur.txt'), et créant un fichier de nom `nom` dans le répertoire courant, et y écrivant l'univers. Chaque ligne de l'univers sera associée à une ligne du fichier, on écrira les caractères 'T' et 'F' à la place de `True` et `False`, et on séparera les entrées par des points-virgules.

Par exemple, si `U` vaut

```
[[True, False, False, False], [False, False, True, True], [False, False, False,
False], [True, False, True, False]]
```

les lignes du fichier seront :

```
T;F;F;F
F;F;T;T
F;F;F;F
T;F;T;F
```

- Q5.** Définir inversement une fonction `charge(nom)`, prenant en entrée une chaîne de caractères désignant un fichier représentant un univers au format précédent et stocké dans le répertoire courant, et renvoyant une liste de listes de booléens décrivant cet univers.

A. Évolution de l'univers

La fonction suivante, renvoyant un univers de taille $N \times N$ contenant uniquement des cases mortes, pourra être utilisée dans la suite, si nécessaire.

```
def univers_mort(N):
    L = []
    for i in range(N):
        L.append([False] * N)
    return L
```

Q6. On rappelle qu'un appel à la fonction `random`, sans argument, supposée importée, renvoie un flottant aléatoire de l'intervalle $[0, 1[$.

Écrire une fonction `init(N)` prenant en entrée un entier strictement positif, et renvoyant un univers $N \times N$ (sous forme de liste de listes), chaque case contenant une cellule vivante avec probabilité $1/3$. Par exemple :

```
>>> U = init(4) ; U
[[True, False, False, False], [False, False, True, True], [False, False, False,
False], [True, False, True, False]]
```

Q7. Écrire une fonction `nb_cellules_vivantes(U)` prenant en entrée un univers `U` et comptant le nombre de cellules vivantes dans l'univers. Cette fonction ne sera pas utilisée dans la suite.

```
>>> nombre_cellules_vivantes(U) #U est la liste prise en exemple en question 1
5
```

Q8. Écrire une fonction `voisins(i, j, N)` prenant en entrée 3 entiers, i, j et N , avec $0 \leq i < N$ et $0 \leq j < N$, et renvoyant la liste des listes de taille 2 d'indices des 8 voisins de (i, j) dans un univers $N \times N$.

Remarque : on fera usage du modulo `%` pour ne pas avoir à traiter à part le cas des cases d'un bord. Par exemple :

```
>>> voisins(2,3,4)
[[1, 2], [1, 3], [1, 0], [2, 2], [2, 0], [3, 2], [3, 3], [3, 0]]
```

Votre fonction pourra renvoyer ces listes de taille 2 dans un ordre quelconque.

Q9. Écrire une fonction `evolve(U)` qui prend en argument un univers à un instant t , et qui renvoie un nouvel univers, résultat de l'évolution de `U` entre les instants t et $t + 1$.

Attention : vous ne devez pas modifier `U`.

```
>>> evolve(U)
[[True, False, True, False], [False, False, False, True], [False, True, True,
False],
[False, True, False, True]]
```

Q10. Quelle est la complexité de `evolve(U)`, en fonction de N , la taille de `U` ?

B. Période et temps d'attraction

Q11. Combien y a-t-il d'univers possibles (c'est-à-dire de grilles différentes) en taille $N \times N$?

À N fixé, l'ensemble des univers possibles est gros, mais fini. Il s'en suit que lorsqu'on fait évoluer l'univers, à partir d'un certain temps on retombe sur un état déjà atteint. À partir de ce moment, l'évolution est périodique. Autrement dit, en notant U_0 l'univers initial, il existe un plus petit r tel que l'univers U_r réapparaîtra, et un plus petit $p > 0$ tel que $U_r = U_{r+p}$, qu'on appelle respectivement temps d'attraction et période (voir figure 3).

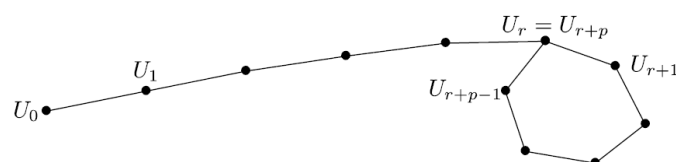


FIGURE 3: Évolution de l'univers : période p et temps d'attraction r

On souhaite calculer temps d'attraction et période dans la suite. La situation se généralise comme suit : on se donne un ensemble fini S et une fonction $f : S \rightarrow S$. Un exemple de telle fonction avec $S = \llbracket 0, 8 \rrbracket$ est donné figure 4, avec le graphe de f représenté également (un autre exemple est la fonction `evolue`, et S l'ensemble des univers possibles en taille fixée).

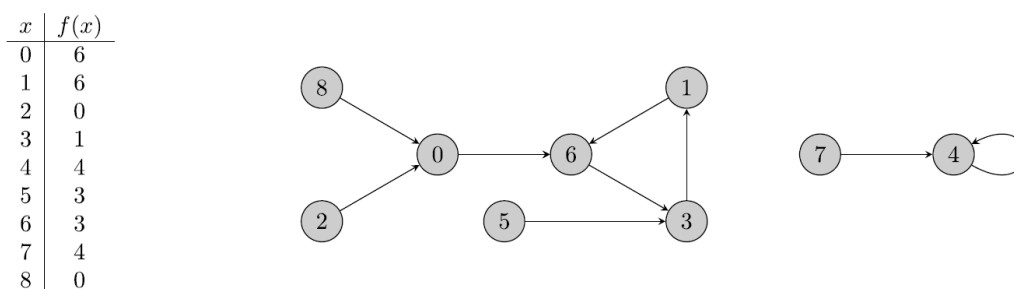


FIGURE 4: Une fonction d'un ensemble fini vers lui-même et sa représentation

On se donne de plus un élément u_0 de S quelconque. On considère la suite $(u_n)_{n \in \mathbb{N}}$ définie par l'application répétée de f , c'est-à-dire $u_{n+1} = f(u_n)$ pour tout $n \geq 0$. Comme l'ensemble S est fini, la suite ne peut prendre une infinité de valeurs distinctes : à partir d'un certain rang, celle-ci est donc périodique. On note dans la suite r le plus petit indice à partir duquel la suite est périodique, et p sa période. Plus précisément, r et p sont définis comme suit :

$$r = \min\{n \geq 0 \mid \exists k > n, u_k = u_n\} \quad \text{et} \quad p = \min\{k > 0 \mid u_{r+k} = u_r\}$$

Avec f la fonction de la figure 4, on a par exemple :

- si $u_0 = 7$, les valeurs de la suite sont 7, 4, 4, 4, ..., on a donc $r = p = 1$;
- si $u_0 = 8$, les valeurs de la suite sont 8, 0, 6, 3, 1, 6, 3, 1, 6, ..., on a donc $r = 2$ et $p = 3$;
- si $u_0 = 1$, les valeurs de la suite sont 1, 6, 3, 1, 6, ..., on a donc $r = 0$ et $p = 3$.

Q12. 1. Écrire une fonction `indice(L,x)` prenant en entrée une liste L et un élément x , et renvoyant le plus petit indice i tel que $L[i] = x$ s'il existe, -1 sinon.

```
>>> indice([3, 4, 1, 0], 3)
0
>>> indice([3, 4, 1, 0], 2)
-1
```

2. Donner sa complexité en fonction de la taille n de la liste, en supposant que le test d'égalité s'effectue en temps constant.

Q13. Écrire une fonction `rang_periode(f,u0)` prenant en entrée une fonction $f : S \rightarrow S$ où S est un ensemble fini, $u_0 \in S$, et renvoyant une liste contenant les entiers r et p définis défini ci-dessus. *Attention* : l'ensemble S n'est pas précisé : on sait simplement que f est bien une fonction d'un ensemble fini vers lui-même. On pourra stocker des termes de la suite dans une liste, et utiliser la fonction précédente. Avec f la fonction de l'exemple, on a :

```
>>> rang_periode(f,7)
[1, 1]
>>> rang_periode(f,8)
[2, 3]
>>> rang_periode(f,1)
[0, 3]
```

Q14. Estimer la complexité de `rang_periode(f,u0)`, en fonction des entiers r et p calculés. On suppose qu'un appel à la fonction f s'effectue avec une complexité constante, de même que le test d'égalité entre deux éléments de S .

C. Calcul efficace : algorithme de Floyd

Pour le jeu de la vie, l'approche précédente demande de stocker en mémoire beaucoup d'univers, si bien qu'elle est inapplicable en pratique dès que N est grand. De plus, les appels à `evolue` et le test d'égalité sur des univers ont une complexité qui dépend également de l'entier N .

On cherche à diminuer à la fois la complexité de la fonction précédente, ainsi que l'espace mémoire utilisé. On propose donc une autre approche. Voici le squelette d'une fonction calculant les entiers r et p , que l'on va compléter.

```
def rang_periode(f,u0):
    u=u0
    v=u0
    t=0
    """ stocke dans t le plus petit élément non nul de A, dans u la valeur u_t et v la valeur v_t """
    while [... à compléter (question 11) ...]:
        u=f(u)
        v=f(f(v))
        t+=1
    """ Ici, t est le plus petit élément non nul de A, u contient u_t, v contient v_t. On calcule p """
    [... à compléter (question 12) ...]
    """ Calcul de r """
    u=u0
    w=u0
    [... à compléter (question 13.1) ...]
    """ Ici on a w=u_p, u=u_0 """
    [... à compléter (question 13.2) ...]
    return [r,p]
```

Q15. Soit $(v_n)_{n \in \mathbb{N}}$ la suite définie par $v_n = u_{2n}$ pour tout $n \in \mathbb{N}$. Montrer que l'ensemble $A = \{n \in \mathbb{N} \mid u_n = v_n\}$ est constitué de l'entier 0 et des multiples de p supérieurs ou égaux à r .

Dans la suite, vous avez **interdiction** d'utiliser des listes : le but est de ne stocker qu'un nombre fini d'éléments de S .

Q16. On note $t = \min(A \setminus \{0\})$. On remarque que $v_0 = u_0$ et $v_{n+1} = f(f(v_n))$ pour tout $n \geq 0$. Compléter la condition de la boucle `while` du script pour que celle-ci permette de stocker dans la variable `t` la valeur t précédemment définie, ainsi que les valeurs u_t dans `u` et v_t dans `v`.

Q17. Pour calculer la période p , il suffit d'itérer f sur u_t jusqu'à retomber sur $u_t = u_{t+p}$. Indiquer comment compléter le code permettant de stocker p dans la variable `p`.

Q18. Considérons la suite $(w_n)_{n \in \mathbb{N}}$ définie par $w_n = u_{n+p}$ pour tout $n \in \mathbb{N}$. On admet que le premier indice k tel que $w_k = u_k$ est précisément r .

1. Compléter le code (question 13.1) pour stocker dans la variable `w` la valeur $w_0 = u_p$.
2. Enfin, compléter le code (question 13.2), pour stocker dans la variable `r` la valeur r définie dans l'énoncé.

Q19. Que pouvez-vous dire de la complexité de cette deuxième version, en fonction des entiers r et p calculés (on supposera ici qu'un appel à f se fait en complexité constante, de même que le test d'égalité de deux éléments de S) ? Justifier.

FIN DU SUJET DU DS01

Ce qui suit n'est pas à traiter pour le DS01.

D. Les univers ayant les plus longues périodes

À l'aide des fonctions des sections précédentes, on a pu générer des univers aléatoires, et calculer les périodes des suites démarrant par ces univers. Ces paires sont stockées sous la forme de listes à 2 éléments `[U, p]` où `U` est un univers, et `p` est la période de la suite de premier terme `U`. On a construit une liste de telles listes de taille 2. On souhaite la trier par période décroissante. On veut faire usage du tri fusion, on a pour cela écrit la fonction suivante :

```
def tri(L):
    """ Tri d'une liste non vide de listes [U,p] """
    if len(L)==1:
        return [L[0]]
    else:
        L1, L2 = separation(L)
        return fusion(tri(L1), tri(L2))
```

Question 15. Écrire la fonction `separation(L)` prenant en entrée une liste `L` ayant au moins deux éléments, et renvoyant une liste `[L1, L2]` avec `L1` et `L2` de même taille à un élément près, telles que les éléments de `L` soient répartis dans `L1` et `L2`.

Question 16. Écrire la fonction `fusion(L1, L2)`, prenant en entrée deux listes contenant des éléments de la forme `[U, p]`, supposées triées par période décroissante, et renvoyant une liste contenant tous les éléments de `L1` et `L2`, triée par période décroissante. On impose une complexité linéaire en la somme des tailles de `L1` et `L2`.

Question 17. Quelle est alors la complexité de `tri(L)` ?

E. SQL

On a sélectionné certains univers menant à des situations intéressantes, et regroupé ceux-ci dans une base de données à deux tables.

- La table **univers** comporte quatre champs de type entier, et un champ de type chaîne de caractères.
 - **id** : identifiant l'univers, c'est la clé primaire de la table ;
 - **n** : l'univers est de taille $n \times n$;
 - **p** et **r** : période et rang d'attraction de la suite de premier terme l'univers ;
 - **nom** : un petit nom donné à l'univers, comme 'planeur' ou 'canon'.
- La table **cellules** comporte également trois champs de type entier :
 - **idu** : un identifiant d'univers, c'est une clé étrangère qui référence le champ **id** de **univers** ;
 - **i** et **j** : servent à référencer les cellules vivantes de l'univers.

Ces tables sont telles qu'un univers d'identifiant **id** possède en (i, j) une cellule vivante si et seulement si (id, i, j) est dans la table **cellules**.

Question 18. On demande de répondre à chacune des questions suivantes par une unique requête SQL.

1. Donner le nombre d'entrées dans la table **univers**.
2. Donner la liste des couples (i, j) d'indices de cellules vivantes de l'univers appelé 'canon'. On suppose qu'un tel univers est présent dans la table **univers** et que ce nom est unique.
3. Donner, pour chaque période présente dans **univers**, le nombre d'univers de la table ayant cette période.
4. Donner la liste des couples d'identifiants d'univers différents de même période et même rang d'attraction.

ANNEXE

Nom :

Prénom :

Classe :


ANNEXE (exercice 1)


Compléter les entêtes « État de la ... », par le mot qui convient, File ou Pile.

Dans le cas d'une file, son entrée sera représentée à gauche, et sa sortie à droite.

Dans le cas d'une pile, sa base sera représentée à gauche, et son sommet à droite.

La colonne de droite de chaque tableau sera intitulée de façon à ce que son contenu puisse être correctement interprété.

	PARCOURS EN LARGEUR du graphe G_8		
	Sommet visité	État de la
Initialement		[]	
à la fin de la 1ère itération			
à la fin de la 2ème itération			
à la fin de la 3ème itération			
...			

	PARCOURS EN PROFONDEUR du graphe G_8		
	Sommet visité	État de la
Initialement		[]	
à la fin de la 1ère itération			
à la fin de la 2ème itération			
à la fin de la 3ème itération			
...			