**Interrogation d'informatique n°10.** *Barème sur 23 pts*

## Exercice A. Base de données et SQL

On considère trois tables

- *Corpus* donnant une liste d'œuvres littéraires

- *Caracteres* donnant pour chaque caractère son symbole et code ASCII

- *Frequences* donnant pour chaque couple (*livre, caractère*) le nombre d'occurrences du caractère dans le texte.

*Corpus*

| *id* | *auteur* | *livre* |
|---|---|---|
| 1 | 'Hugo' | 'Notre-Dame de Paris' |
| 2 | 'Vian' | 'L'écume des jours' |
| ⋮ | ⋮ | ⋮ |

*Caracteres*

| *symbole* | *code* |
|---|---|
| 'a' | 97 |
| 'b' | 98 |
| ⋮ | ⋮ |

*Frequences*

| *id* | *code* | *nombre* |
|---|---|---|
| 1 | 97 | 25100 |
| 1 | 98 | 28751 |
| ⋮ | ⋮ | |

**1)** [1 pt] Écrire une requête SQL renvoyant la liste (sans doublon) des auteurs apparaissant dans *Corpus*.

**2)** [2 pts] Écrire une requête SQL renvoyant la table donnant pour chaque auteur le nombre de caractères utilisés dans le corpus total. Autrement dit, la table demandée admet deux attributs correspondant respectivement à l'auteur et au nombre de caractéres de son corpus.

On classera les données par valeur décroissante du nombre de caractères.

**3)** [2 pts] Écrire une requête renvoyant les deux symboles de lettres dont les fréquences dans les œuvres du centralien Boris Vian sont les plus grandes.

## Exercice B. Arbres couvrants aléatoires et labyrinthes

### Partie I. Génération de permutations aléatoires par la méthode de Knuth

La fonction `numpy.random.randint(a,b)` renvoie un entier aléatoire compris entre $a$ et $b-1$.

On veut générer une permutation aléatoire d'une liste $L$ d'entiers distincts.
On veut implémenter l'algorithme de Knuth : cet algorithme effectue $n$ échanges :

Pour $0 \leq i < n$, on permute `L[i]` avec `L[j]`, où $j$ est un entier aléatoire compris entre 0 et $i$ inclus.

Par exemple, `permut([1,4,5])` renvoie une des listes `[1,4,5]`, `[1,5,4]`, `[4,1,5]`, `[4,5,1]`, `[5,1,4]` ou `[6,4,1]`.

**1)** [1.5 pt] Écrire une fonction `permut(L)` qui renvoie une permutation aléatoire de $L$ par cet algorithme. On utilisera `L.copy()` pour éviter de modifier $L$.

**Partie II. Parcours en profondeur aléatoire d'un graphe**

On considère un graphe connexe $G' = (S, A)$ non orienté défini par la liste `G` des listes d'adjacence :

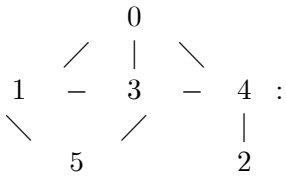
- L'ensemble des sommets est $S = \{0, 1, 2, ..., n-1\}$

- Pour tout $i \in S$, `G[i]` est la liste des voisins de $i$ (sans doublons et classés par ordre croissant).

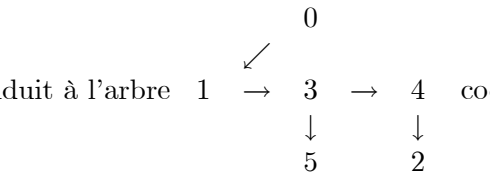
Un sous-graphe de $G$ est un graphe $G' = (S, A')$, avec $A' \subset A$.

On rappelle que le parcours en profondeur consiste à visiter successivement les sous-arbres des voisins (non encore visités) d'un sommet en considérant ces voisins dans l'ordre donné par les listes d'adjacence. Par exemple, si 1, 3 et 4 sont des sommets voisins du sommet 0, un parcours en profondeur commencera par traiter le sommet 1, avant de traiter les sommets 3 et 4.

Un parcours en profondeur issu du sommet 0 permet de construire un arbre qui est un sous-graphe de $G$ : On code cet arbre par le tableau $P$ des pères en associant à chaque sommet son prédécesseur dans le parcours en profondeur, et en attribuant 0 au sommet 0 qui est la racine de l'arbre.

Par exemple, le parcours en profondeur du graphe $G$ : 

:

conduit à l'arbre 

codé par le tableau des pères $T = [0, 0, 4, 1, 3, 3]$.

**2)** [1 pt] Donner une condition nécessaire et suffisante sur $G$ pour que $T[1] = T[2] = 0$.

**3)** [4.5 pts] On suppose désormais connue une fonction `permut` qui étant donnée une liste $L$ d'éléments distincts renvoie une permutation aléatoire de ses éléments.

**Un parcours en profondeur aléatoire** se fait en utilisant l'algorithme suivant :

- On part du sommet 0

- Le traitement d'un sommet $s$ consiste à permuter la liste de ses voisins, à parcourir la liste des voisins dans l'ordre ainsi défini, et pour chaque sommet $t \in G$ non encore traité, on ajoute l'arête $(s, t)$ à l'arbre et on traite $t$.

Écrire une fonction `parcours_alea(G)` qui renvoie l'arbre obtenu $T$ en utilisant un parcours aléatoire de $G$ issu du sommet `0`.

**Attention** : On proposera deux méthodes :

- **une** utilisant une fonction auxiliaire récursive `traite`

- **l'autre** utilisant une `pile` pour stocker les sommets à traiter.

**Dans les deux cas**, on utilisera un tableau booléen `visites` pour marquer les sommets déjà traités.

**Partie III. Création de labyrinthes**

On considère un maillage carré $N \times N$ dont les cellules sont indexées par $(i, j) \in [\![0, N-1]\!]^2$ :

Exemple pour $N = 3$ :

| | | |
|---|---|---|
| $(0,2)$ | $(1,2)$ | $(2,2)$ |
| $(0,1)$ | $(1,1)$ | $(2,1)$ |
| $(0,0)$ | $(1,0)$ | $(2,0)$ |

Deux cellules d'indices resepctifs $(i, j)$ et $(k, l)$ sont voisines ssi $|i-k| + |j-l| = 1$.

**4) Afin de se ramener à un graphe dont les sommets sont étiquetés par des entiers, on associe à chaque couple $(i, j)$ l'entier $k = i + j\ N$.**

a) [0.5 pt] Écrire une fonction `code(c,N)` qui étant donné un couple $c = (i, j)$ renvoie $k = i\ + j\ N$.

b) [1 pt] Écrire une fonction `decode(k,N)` qui étant donné un entier $k \in [\![0, N^2 - 1]\!]$ renvoie le couple $(i, j) \in [\![0, N-1]\!]^2$ tel que $k = i\ + j\ N$.

**5)** a) [1.5 pt] Écrire une fonction `voisins(c,N)` qui étant donné un couple $c = (i, j) \in [\![0, N-1]\!]^2$ renvoie la liste des voisins $c' = (k, l)$ de $(i, j)$ dans le maillage à $N^2$ sommets.

Par exemple, `voisins((1,0),3)` renvoie `[(0,0),(2,0),(1,1)]`.

b) [2.5 pts] Écrire une fonction `graphe_lab(N)` qui renvoie le graphe non orienté dont

- les $N^2$ sommets $(i, j) \in [\![0, N-1]\!]^2$ codés par des entiers de 0 à $n = N^2 - 1$
- les voisins de $(i, j)$ sont les $(k, l) \in [\![0, N-1]\!]^2$ tels que $|i-k| + |j-l| = 1$.

Exemple pour $N = 3$ : Le maillage peut être représenté par $N^2$ cellules : on obtient

| | | |
|---|---|---|
| $(0,2)$ | $(1,2)$ | $(2,2)$ |
| $(0,1)$ | $(1,1)$ | $(2,1)$ |
| $(0,0)$ | $(1,0)$ | $(2,0)$ |

, c'est-à-dire le graphe

$$\begin{array}{ccccc} 6 & \leftrightarrow & 7 & \leftrightarrow & 8 \\ \updownarrow & & \updownarrow & & \updownarrow \\ 3 & \leftrightarrow & 4 & \leftrightarrow & 5 \\ \updownarrow & & \updownarrow & & \updownarrow \\ 0 & \leftrightarrow & 1 & \leftrightarrow & 2 \end{array}$$

En PYTHON, le graphe associé au maillage est donc codé par la liste de listes

```
G = [[1,3],[0,2,4],[1,5],[0,4,6],[1,3,6,7],[2,4,8],[3,7],[4,6,8],[5,7]]
```

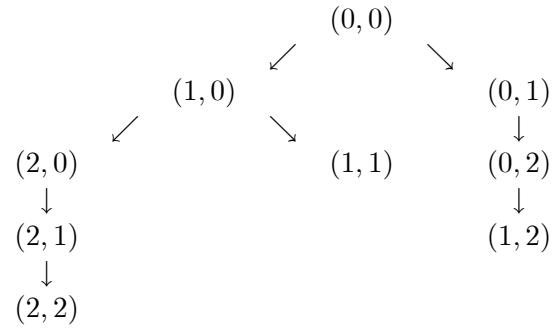
**6)** Chaque cellule $(i, j)$ du maillage correspond au carré du plan $[i, i+1] \times [j, j+1]$.
On considère que chaque cellule $(i, j)$ du maillage est entouré de 4 murs. Les murs sont horizontaux ou verticaux, d'abscisse ou d'ordonnée entière appartenant à $[\![0, N]\!]$.

Il s'agit donc dans $\mathbb{R}^2$ de la réunion

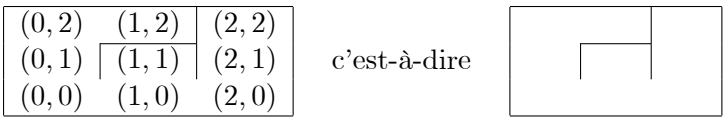
- des segments reliant $(i, j)$ et $(i+1, j)$, avec $(i, j) \in [\![0, N-1]\!] \times [\![0, N]\!]$
- et des segments reliant $(i, j)$ et $(i, j+1)$, avec $(i, j) \in [\![0, N]\!] \times [\![0, N-1]\!]$.

On considère l'arbre $T$ associé à un parcours en profondeur aléatoire du graphe associé au maillage, où le parcours en profondeur part du sommet initial $(0, 0)$. **On obtient un labyrinthe en supprimant les murs situés entre deux cellules qui sont père et fils dans l'arbre $T$.**

Par exemple, si $T$ est défini par




le labyrinthe associé est




c'est-à-dire



a) [2.5 pts] Écrire une fonction `murs_lab(N)` qui crée un arbre aléatoire dans le graphe associé au maillage et renvoie le dictionnaire `D` définissant le graphe au labyrinthe :

- les clés du dictionnaire sont les couples $(i, j) \in [\![0, N-1]\!]^2$

- la valeur `D[(i,j)]` est la liste des cellules voisines dans le labyrinthe.

Dans l'exemple ci-dessus, le dictionnaire `D` est défini par

```
D[(0,0)] = [(1,0),(0,1)]
D[(1,0)] = [(0,0),(2,1),(1,1)]
D[(2,0)] = [(1,0),(2,1)]
D[(2,1)] = [(2,0),(2,2)]
D[(2,2)] = [(2,1)]
D[(1,1)] = [(1,0)]
D[(0,1)] = [(0,0),(0,2)]
D[(0,2)] = [(0,1),(1,2)]
D[(1,2)] = [(0,2)]
```

b) [2.5 pts] On suppose que la bibliothèque graphique est chargée :

```
import matplotlby.pyplot as plt
```

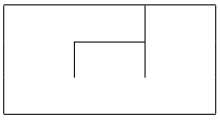
On rappelle que le tracé du segment d'extrémités $(x, y)$ et $(x', y')$ s'obtient par :

```
plt.plot([x,x'],[y,y'])
```

On superpose les graphes et on affiche la fenêtre graphique par des instructions de la forme

```
plt.plot(graphe1) ; plt.plot(graphe2) ; ... ; plt.plot(grapheM) ; plt.show()
```

Écrire une fonction `dessin_lab(N)` qui affiche un labyrinthe aléatoire à $N^2$ cellules, c'est-à-dire un schéma de la forme



**Attention** : Bien noter que le mur horizontal reliant les points $(i, j)$ à $(i+1, j)$ sépare les deux cellules d'indices $(i, j)$ et $(i, j-1)$, lorsque $0 < j < N$.

**Partie IV. Centre du labyrinthe**

Dans un graphe connexe non orienté :

- une extrémité est un sommet admettant un unique voisin

- la distance entre deux sommets est la longueur minimale d'un chemin reliant les deux sommets

- l'excentricité d'un sommet est la distance maximale de ce sommet aux autres sommets

- un sommet est dit central ssi il est d'excentricité mimimale

- le centre est l'ensemble des sommets centraux.

Par exemple, dans le labyrinthe défini par $\begin{array}{|ccc|}\hline (0,2) & (1,2) & (2,2) \\ (0,1) & (1,1) & (2,1) \\ (0,0) & (1,0) & (2,0) \\ \hline\end{array}$,

- les extrémités du labyrinthe sont $(1,2)$, $(1,1)$ et $(2,2)$

- le sommet $(1,2)$ est d'excentricité 7 associée au chemin de longueur 7 :

$$(1,2) \to (0,2) \to (0,1) \to (0,0) \to (1,0) \to (2,0) \to (2,1) \to (2,2)$$

- le centre est formé des deux cellules $(0,0)$ et $(1,0)$ : toute distance aux autres cellules est $\leq 4$.

**On admet la propriété suivante** : étant donné un graphe connexe $G$ de cardinal $\geq 3$, le graphe $G'$ obtenu en supprimant dans $G$ ses extrémités (et les arêtes associées) est connexe et a le même centre que $G$. De plus, $G'$ est un arbre lorsque $G$ est un arbre (c'est-à-dire connexe acyclique).

*Remarque* : Le graphe d'un labyrinthe est un arbre, donc il contient au moins une extrémité, et de ce fait le centre d'un labyrinthe contient une ou deux cellules.

**7)** [3 pts] Écrire une fonction `centre(D)` qui étant donné le dictionnaire du graphe associé à un labyrinthe (cf question 6) a)), renvoie le centre du labyrinthe (sous forme d'une liste des cellules centrales).

**Attention** : Afin d'avoir un algorithme de complexité linéaire, on utilisera :

- le dictionnaire $D$ (qui ne sera pas modifié) défini à la question 6) a)

- un dictionnaire $M$ dont les clés sont les cellules non encore supprimées et dont les valeurs sont les nombres de cellules voisines non encore supprimées

- une liste $L$ des clés de $M$ dont la valeur vaut 1 (ce sont les extrémités du graphe associé à $M$)

On rappelle qu'on supprime un élément $c$ d'un dictionnaire $M$ par l'instruction : `del M[c]`

**Complément culturel : Utilisation de la structure d'union-find**

En considérant les opérations usuelles **union** et **find** sur les partitions, on obtient une autre méthode efficace permettant de construire un sous-arbre aléatoire d'un graphe connexe.