

## Correction CB - info

**Exercice 1.** On considère une base de données relationnelle concernant des gènes et leur expression dans différents tissus chez plusieurs espèces.

Schéma relationnel

- **Gène**(*id\_gène*, nom\_gène, chromosome, position\_début, position\_fin, nombre\_d\_exons)
- **Expression**(*id\_gène*, tissu)
- **Espèce**(*id\_espèce*, nom\_espèce)
- **GèneEspèce**(*id\_gène*, *id\_espèce*)

Ecrire des requêtes SQL qui permettent de :

1. Donner les noms des gènes situés sur le chromosome 1.
2. Donner les noms gènes qui possèdent plus de 10 exons et sont exprimés dans le tissu « foie ».
3. Donner le nombre moyen d'exons des gènes qui s'exprime dans le tissu « foie ».
4. Donner les noms des espèces dans lesquelles on retrouve le gène nommé « TP53 ».
5. Donner le nombre de gènes pour chaque tissu.

### Correction 1.

---

```
1 SELECT nom_gene FROM Gène WHERE chromosome=1
```

---

---

```
1 SELECT nom_gene
2 FROM Gène JOIN Expression
3 ON Gène.id_gène = Expression.id_gène
4 WHERE Expression.tissu='foie' AND nombre_d_exons>9
```

---

---

```
1 SELECT AVG(nombre_d_exons)
2 FROM Gène JOIN Expression
3 ON Gène.id_gène = Expression.id_gène
4 WHERE Expression.tissu='foie'
```

---

---

```
1 SELECT nom_espece
2 FROM Espèce JOIN Gène JOIN GèneEspèce
3 ON Gène.id_gène = GèneEspèce.id_gène and Espèce.id_espèce=GèneEspèce.id_epece
4 WHERE Gène.nom_gène='TP53'
```

---

---

```
1 SELECT COUNT(id_gène)
2 FROM Gène JOIN Expression
3 ON Gène.id_gène= Expression.id_gène
4 GROUP BY tissu
```

---

**Exercice 2.** 1. Donner le résultat des expressions suivantes :

---

```
1 len("AGTCAGT")
2 "AGT"[1]
3 "ATCG" + "AAGC"
4 "ATGC" * 2
```

---

On souhaite comparer deux séquences d'ADN de même longueur, pour cela on va comparer base par base et compter le nombre de bases qui diffèrent d'un brin à un autre.

2. **Écrire une fonction** `distance(seq1, seq2)` qui prend en argument deux chaînes de caractères de même taille et qui retourne le nombre de caractères qui diffèrent d'une chaîne à l'autre.

Le programme affichera un message d'erreur si les chaînes ne sont pas de même taille.

Exemple :

---

```
1 >>> distance("ATGC", "AGGT")
2 2
```

---

(Ici les bases T/G et C/T diffèrent à l'indice 1 et 3)

3. On dispose de la fonction suivante :

---

```
1 def mystere(s1, s2):
2     result = []
3     if len(s1)!=len(s2):
4         return(False)
5     for x in range(len(s1)):
6         if s1[x]!=s2[x]:
7             result.append(x)
8     return result
```

---

(a) Que retourne cette fonction si on l'appelle avec `s1 = "ATGC"` et `s2 = "AGGT"` ?

(b) Comment retrouver le résultat de la fonction `distance` à l'aide de la fonction `mystere` ?

## Correction 2.

---

```
1. len("AGTCAGT")
2 >>> 7
3 "AGT"[1]
4 >>> G
5 "ATCG" + "AAGC"
6 >>> "ATCGAAGC"
7 "ATGC" * 2
8 >>> "ATCGATGC"
```

---

---

```
2. def distance(s1, s2):
2     result = 0
3     if len(s1)!=len(s2):
4         return('erreur')
5     for x in range(len(s1)):
6         if s1[x]!=s2[x]:
7             result+=1
8     return result
```

---

---

3. (a) `>>> mystere("ATGC", "AGGT")`  
2 `[1,3]`

---

(b) `len(mystere(s1, s2))`  
2

---

renvoie la même valeur que `distance(s1,s2)`

- Exercice 3.**
1. Écrire une fonction `creation` qui prend en argument deux entiers  $n$  et  $m$ , et retourne un tableau `numpy` de taille  $(n, m)$  rempli de valeurs entières choisies aléatoirement entre 1 et 9 (inclus), selon une distribution uniforme.
  2. Écrire une fonction `transposee` qui prend en argument un tableau `numpy` représentant une matrice et retourne sa transposée. On n'utilisera pas la fonction `np.transpose`.
  3. Écrire une fonction `symmetrique` qui prend en argument une matrice `numpy`, et retourne `True` si la matrice est symétrique, `False` sinon.
  4. On s'intéresse maintenant à la validation d'une grille de Sudoku.

**Rappel :** Une grille de Sudoku est une matrice  $9 \times 9$  composée uniquement d'entiers de 1 à 9. Elle est dite *valide* si elle respecte simultanément les trois conditions suivantes :

- chaque **ligne** contient tous les entiers de 1 à 9, sans répétition ;
- chaque **colonne** contient tous les entiers de 1 à 9, sans répétition ;
- chaque **bloc**  $3 \times 3$  (il y en a 9) contient tous les entiers de 1 à 9, sans répétition.

1	2	3
4	5	6
7	8	9

FIGURE 1 – Représentation des blocs  $3 \times 3$  d'une grille de Sudoku

- (a) Écrire une fonction `contient_tous_les_nombres` qui prend en argument une liste, et retourne `True` si cette liste contient tous les entiers de 1 à 9 *une et une seule fois*, `False` sinon.
- (b) Écrire une fonction `sudoku_ligne` qui prend en argument une matrice `numpy` de taille  $9 \times 9$ , et retourne `True` si chaque ligne contient exactement tous les entiers de 1 à 9, `False` sinon.
- (c) Écrire une fonction `sudoku_colonne` qui prend en argument une matrice `numpy` de taille  $9 \times 9$ , et retourne `True` si chaque colonne contient exactement tous les entiers de 1 à 9, `False` sinon.
- (d) Trois fonctions sont proposées ci-dessous pour vérifier si chaque bloc  $3 \times 3$  contient tous les nombres de 1 à 9. Une seule est correcte. Indiquer laquelle, et expliquer brièvement pourquoi les deux autres ne conviennent pas.

---

```

1 def sudoku_bloc1(tableau):
2     n, m = np.shape(tableau)
3     if n != 9 or m != 9:
4         return False
5     for i in range(3): # lignes de blocs
6         for j in range(3): # colonnes de blocs
7             sous_tableau = []
8             for k in range(3):
9                 for l in range(3):
10                     row = 3 * i + k
11                     col = 3 * j + l
12                     sous_tableau.append(tableau[row, col])
13             if contient_tous_les_nombres(sous_tableau) == False:
14                 return False
15     return True

```

---

```

1 def sudoku_bloc2(tableau):
2     n, m = np.shape(tableau)
3     if n != 9 or m != 9:
4         return False
5     for i in range(3): # lignes de blocs
6         for j in range(3): # colonnes de blocs
7             sous_tableau = []
8             for k in range(3):
9                 for l in range(3):
10                     row = 3 * i + k
11                     col = 3 * j + l
12                     sous_tableau.append(tableau[row, col])
13             if contient_tous_les_nombres(sous_tableau) == True:
14                 return True
15     return False

```

---

```

1 def sudoku_bloc3(tableau):
2     n, m = np.shape(tableau)
3     if n != 9 or m != 9:
4         return False
5     for i in range(3): # lignes de blocs
6         for j in range(3): # colonnes de blocs
7             sous_tableau = []
8             for k in range(3):
9                 for l in range(3):
10                     row = i + 3 * k
11                     col = j + 3 * l
12                     sous_tableau.append(tableau[row, col])
13             if contient_tous_les_nombres(sous_tableau) == False:
14                 return False
15     return True

```

---

- (e) En utilisant les fonctions précédentes, écrire une fonction `sudoku` qui prend en argument une matrice `numpy` de taille  $9 \times 9$ , et retourne `True` si cette matrice représente une grille de Sudoku valide et `False` sinon.

### Correction 3.

---

```

1. def creation(n,m):

```

---

```

2     A=np.array((n,m))
3     for i in range(n):
4         for j in range(m)
5             A[i,j]=rd.randint(1,9)
6     return A

```

---

```

12. def transpose(M):
2     n,m=np.shape(M)
3     T=np.array((m,n))
4     for i in range(m):
5         for j in range(n)
6             T[i,j]=M[j,i]
7     return T

```

---

```

13. def symmetrique(M):
2     return transpose(M)==M

```

---

```

4. (a) def contient_tous_les_nombres(L):
2     if len(L)!=9:
3         return False
4     for i in range(1,10):
5         if i not in L:
6             return False
7     return True

```

---

```

(b) def sudoku_ligne(M):
2     for i in range(n):
3         L=[A[i,j] for j in range(9)]
4         if contient_tous_les_nombres(L) == False:
5             return False
6     return True

```

---

```

(c) def sudoku_colonne(M):
2     return sudoku_ligne(transpose(M))

```

---

(d) C'est le programme `sudoku_bloc1` qui fonctionne. `sudoku_bloc2` ne fonctionne pas car retourne vrai en vérifiant le premier bloc sans vérifier les autres. `sudoku_bloc3` ne regarde pas les numéros des blocs.

---

```

(e) def sudoku(M):
2     b1=sudoku_ligne(M)
3     b2=sudoku_colonne(M)
4     b3=sudoku_bloc1(M)
5     return b1 and b2 and b3

```

---