

CONSIGNES GÉNÉRALES



DOCUMENTS NON AUTORISÉS
CE CAHIER D'EXAMEN COMPORTE 22 PAGES + 2 PAGES VIDES
LES RÉPONSES DOIVENT ÊTRE ÉCRITES DANS
LES ESPACES RÉPONSES DÉDIÉES
EN CAS DE BESOIN UTILISER LES PAGES VIDES EN FIN DU CAHIER, DANS
CE CAS, IL FAUT LE SIGNALER DANS LES CASES RÉPONSES ALLOUÉES
L'USAGE DES CALCULATRICES EST INTERDIT
IL FAUT RESPECTER IMPÉRATIVEMENT LES NOTATIONS DE L'ÉNONCÉ
VOUS POUVEZ ÉVENTUELLEMENT UTILISER L'ANNEXE

Le sujet comporte trois parties qui traitent les aspects suivants :

Partie I : Programmation orientée objet (Q1..Q21).

Partie II : Programmation procédurale (Q22..Q25).

Partie III : Base de données relationnelle (Q26..Q36).

L'objectif des parties I et II est d'étudier la performance de quelques **algorithmes d'ordonnement de processus** dans un système d'exploitation. Un **système d'exploitation** est un logiciel qui permet d'exploiter les ressources d'une machine, à titre d'exemples : Windows, Linux, MacOS, Android, etc.

Un **processus** est défini comme étant un programme chargé en mémoire centrale en vue d'être exécuté par le processeur. Un processus est donc né lors du chargement d'un programme et se termine à la fin de l'exécution de ce programme. Un processus peut être à l'un des états suivants :

- **Prêt** : s'il dispose de toutes les ressources nécessaires à son exécution à l'exception du processeur.
- **Elu** : s'il est en cours d'exécution par le processeur.
- **Bloqué** : s'il est en attente d'un événement ou bien d'une ressource pour pouvoir continuer.

Un **ordonnanceur** est un processus important du système d'exploitation qui gère l'allocation du temps processeur. Plusieurs processus peuvent être à l'état "Prêt", dans ce cas l'ordonnanceur se charge de choisir parmi eux celui qui devra être traité en premier lieu selon une **stratégie d'ordonnement**. Un ordonnanceur fait face à deux problèmes principaux :

- Comment élire un processus à exécuter parmi les processus prêts ?
- Durant combien de temps le processeur est alloué au processus élu ?

Partie I : Programmation Orientée Objet

Dans le but de simuler un ordonnanceur de processus, on propose d'implémenter un ensemble de classes permettant de mettre en place certaines tâches de l'ordonnanceur :

- **Classe Processus** : modélise les caractéristiques d'un processus.
- **Classe FileProcessus** : modélise une liste de processus.
- **Classe StrategieOrdonnancement** : encapsule toutes les données et méthodes communes aux différentes stratégies d'ordonnancement (FCFS, SJF et RR). Elle est dédiée à la spécialisation par héritage.
- **Classe FCFS** : modélise la stratégie d'ordonnancement **FCFS** (First-Come First-Served). Elle traite les processus dans l'ordre de leurs soumissions (temps d'arrivée). L'organisation de la file d'attente des processus prêts est donc du type "First In First Out".
- **Classe SJF** : modélise la stratégie d'ordonnancement **SJF** (Shortest Job First). Elle favorise les processus présents ayant une durée d'exécution plus courte.
- **Classe RR** : modélise la stratégie d'ordonnancement **RR** (Round Robin). C'est une stratégie préemptive où le processeur est alloué à chaque processus pour une durée limitée dite quantum. Le processus élu selon l'ordre de sa soumission est exécuté durant ce quantum, puis remis en queue de la file d'attente. Ceci engendre une gestion circulaire de la liste des processus prêts.

Interface de la classe Processus

Attributs

- **nomP**, chaîne de caractères représentant le nom du processus (supposé unique).
- **taP**, entier représentant le temps d'arrivée du processus.
- **deP**, entier représentant la durée totale d'exécution du processus.
- **etatP**, chaîne de caractères représentant l'état du processus.
- **trP**, entier représentant la durée restante pour l'achèvement de l'exécution du processus initialisé par **deP**.

NB : le temps est mesuré en nombre de cycles d'horloge du processeur.

Méthodes à implémenter

Q1. `__init__(...)` : initialise un processus à partir des paramètres **nomP**, **taP**, **deP** et **etatP** représentant respectivement le nom du processus, son temps d'arrivée, sa durée d'exécution et son état.

Exemple 1: L'objet processus **P** de nom "P", ayant un temps d'arrivée 1, une durée nécessaire d'exécution 8 et l'état "Elu" est instancié comme suit :



```
P = Processus ("P", 1, 8, "Elu")
```

Espace de réponse pour Q1

```
class Processus:  
    def __init__(self, nomP, taP, deP, etatP):
```

Q2. `__str__(...)` : représente textuellement un processus (voir l'exemple 2).

Exemple 2: pour le processus P créé dans l'exemple 1



```
>>> str(P)  
"Nom : P - Temps d'Arrivée : 1 - Durée d'Exécution : 8 - Etat : Elu"
```

Espace de réponse pour Q2

```
def __str__(self):
```

Q3. `__eq__(...)` : retourne un booléen indiquant si les deux processus `self` et `other` ont le même nom.

Espace de réponse pour Q3

```
def __eq__(self, other):
```

Q4. `changerEtat(...)` : change l'état courant du processus vers l'état `ns` passé en paramètre.

Espace de réponse pour Q4

```
def changerEtat(self, ns):
```

Interface de la classe FileProcessus

Attributs

- `data`, une liste de processus.

Méthodes à implémenter

Q5. `__init__(...)` : initialise l'attribut `data` à une liste vide.

Espace de réponse pour Q5

```
class FileProcessus:  
    def __init__(self):
```

Q6. `__len__(...)` : retourne le nombre de processus présents dans l'instance actuelle de `FileProcessus`.

Espace de réponse pour Q6

```
def __len__(self):
```

Q7. `__str__()` : permet de représenter un objet de type `FileProcessus` par une chaîne selon le format suivant :

```
"""
Liste de processus : [
Nom : nomP1 - Temps Arrivée : taP1 - Durée d'Exécution : deP1 - Etat : etatP1,
Nom : nomP2 - Temps Arrivée : taP2 - Durée d'Exécution : deP2 - Etat : etatP2,
...
]
"""
```

Espace de réponse pour Q7

```
def __str__(self):
```

Q8. `__contains__()` : reçoit un processus `p` en paramètre et vérifie s'il est présent dans la file actuelle.

Espace de réponse pour Q8

```
def __contains__(self,p):
```

Q9. `__getitem__(...)` : retourne un processus de la file actuelle dont le nom `nomP` est passé en paramètre.

 Espace de réponse pour Q9

```
def __getitem__(self, nomP):
```

Q10. `nomProcessusElu(self)` : retourne le nom du premier processus ayant l'état "Elu" dans la file actuelle, None si aucun processus ne dispose de l'état "Elu".

 Espace de réponse pour Q10

```
def nomProcessusElu(self):
```

Q11. `ajouterProcessus(...)` : ajoute un processus `p` passé en paramètre dans la file actuelle (En faisant la gestion des exceptions nécessaires).

 Espace de réponse pour Q11

```
def ajouterProcessus(self, p):
```

Q12. `retirerProcessus(...)` : retire et retourne de la file actuelle le processus de `nomP` passé en paramètre.

Espace de réponse pour Q12

```
def retirerProcessus(self,nomP):
```

Interface de la classe `StrategieOrdonnancement`

Attributs

- `fp` : instance de la classe `FileProcessus` contenant les processus en cours d'ordonnement.
- `horloge` : un entier représentant l'instant actuel exprimé en cycles d'horloge.
- `stats` : un dictionnaire où les clés sont les noms des processus en cours d'ordonnement. Chaque clé est associée à un tuple contenant des mesures reflétant des critères de performances qui seront détaillés ultérieurement.

Méthodes à implémenter

Q13. `__init__(...)` : initialise l'attribut `fp` à partir du paramètre `fp`, `horloge` à zéro et `stats` par un dictionnaire vide.

Espace de réponse pour Q13

```
class StrategieOrdonnancement:  
    def __init__(self, fp):
```

La méthode `elire(...)` : responsable du choix du processus à exécuter. Cette méthode sera redéfinie dans les classes filles.

```
def elire(self):  
    # pour cette classe le script de cette méthode  
    # n'est pas demandé
```

Q14. exécuter(...) : prend un paramètre *q* ayant par défaut la valeur *None* et exécute le processus élu selon la démarche suivante :

- chercher le nom *nomP* du premier processus ayant l'état "Elu" dans *fp* ;
- si aucun processus de *fp* n'est élu, incrémenter l'horloge d'une unité et interrompre l'exécution actuelle ;
- sinon retirer de *fp* le processus *p* de nom *nomP* et calculer *tep* le temps processeur alloué à son exécution :
- si *q* est *None* affecter à *q* le temps restant pour l'achèvement de *p* ;
- *tep* prend le minimum entre le temps restant du processus *p* et *q* ;
- mettre à jour *horloge* en l'incrémentant par *tep* ;
- mettre à jour le temps restant de l'exécution du processus *p* en le décrémentant par *tep* ;
- si le processus *p* est achevé (le temps restant devient nul) :
 - calculer dans *tr* le temps de réponse du processus *p* (voir équation 1) ;
 - calculer dans *tat* le temps d'attente du processus *p* (voir équation 2) ;
 - insérer le tuple (*tr*, *tat*) dans le dictionnaire *stats* associé à la clé *nomP* du processus *p* ;

NB : Pour un processus P_i quelconque, on a :

$$trP_i = \text{instant actuel de l'horloge} - taP_i$$

Équation 1 – temps de réponse d'un processus P_i

et

$$tat P_i = tr P_i - deP P_i$$

Équation 2 – temps d'attente d'un processus P_i

- si le processus *p* n'est pas encore achevé, changer son état à "Prêt" et le remettre dans *fp*.

Espace de réponse pour Q14

```
def exécuter(self, q = None):
```

☛ Suite espace de réponse pour Q14

Q15. `simuler(...)` : prend en paramètre un entier positif `duree` (durée totale de la simulation) et applique itérativement les deux étapes suivantes :

- choix du processus à exécuter via la méthode `elire` ;
- exécution du processus élu via la méthode `executer` ;

La simulation s'arrête quand l'horloge dépasse `duree` ou bien jusqu'à l'achèvement de l'exécution des processus de la file `fp`.

☛ Espace de réponse pour Q15

```
def simuler(self, duree):
```

Q16. `criteres(...)` : calcule et retourne un tuple contenant le temps de réponse moyen (TRM) et le temps d'attente moyen (TAM) calculés à partir du dictionnaire `stats` en se basant sur les formules suivantes :

TRM : moyenne des temps de réponses des processus achevés :

$$\text{TRM} = \frac{\sum_{i=1}^n \text{tr } P_i}{n} \text{ où } n \text{ est le nombre total de processus achevés.}$$

Équation 3 – Le temps de réponse moyen

TAM : moyenne des temps d'attentes des processus achevés :

$$\text{TAM} = \frac{\sum_{i=1}^n \text{tat } P_i}{n} \text{ où } n \text{ est le nombre total de processus achevés.}$$

Équation 4 – Le temps d'attente moyen

Espace de réponse pour Q16

```
def criteres(self):
```

Interface de la classe fille FCFS

Méthodes à implémenter

Q17. elire(...) : élit le processus ayant le temps d'arrivée minimal parmi les processus ayant un temps d'arrivée inférieur ou égal à l'instant actuel de l'horloge.

Espace de réponse pour Q17

```
class FCFS(StratégieOrdonnancement):  
    def elire(self):
```

Interface de la classe fille SJF

Méthodes à implémenter

Q18. `elire(...)` : élit le processus ayant la plus courte durée d'exécution avec un temps d'arrivée inférieur ou égal au temps courant de l'horloge.

Espace de réponse pour Q18

```
class SJF(StratégieOrdonnancement):  
    def elire(self):
```

Interface de la classe fille RR

Attributs

- `q` : entier positif valeur du quantum alloué aux processus.

Méthodes à implémenter

Q19. `__init__(...)` initialise un attribut d'instance `q` à partir du paramètre `q`.

Espace de réponse pour Q19

```
class RR(StratégieOrdonnancement):  
    def __init__(self, fp, q):
```

Q20. `executer(...)` : exécute le processus élu durant le quantum alloué.

 Espace de réponse pour Q20

```
def executer(self):
```

Q21. `elire(...)` : élit le premier processus de la file ayant un temps d'arrivée inférieur ou égal au temps courant de l'horloge.

 Espace de réponse pour Q21

```
def elire(self):
```

Partie II : Programmation Procédurale

Dans cette partie on s'intéresse à simuler et à comparer les différentes stratégies d'ordonnancement appliquées à une file de processus supposés stockés dans un fichier texte où chaque ligne est de la forme :

"nomP#taP#deP#etatP\n"



```
A#0#3#Prêt
B#1#8#Prêt
C#2#6#Prêt
D#3#4#Bloqué
E#4#4#Prêt
F#6#5#Prêt
G#8#2#Prêt
H#9#1#Prêt
```

FIGURE 1 – Extrait d'un fichier texte décrivant une file de processus

Pour un échantillon donné de processus, un algorithme d'ordonnancement est jugé performant sur la base de la comparaison de deux critères, le temps de réponse moyen (TRM) en premier lieu et le temps d'attente moyen (TAM) en second lieu.

Q22. Écrire une fonction nommée `processusTxt` qui prend en entrée une ligne contenant les informations d'un processus comme illustré par la FIGURE 1 et retourne une instance de la classe `Processus`.

 Espace de réponse pour Q22

```
def processusTxt(ligne):
```

Q23. Écrire une fonction nommée `chargement` qui prend en entrée `nomF` le nom d'un fichier texte (comme indiqué sur la figure 1) et retourne une instance de la classe `FileProcessus` contenant tous les processus du fichier.

 Espace de réponse pour Q23

```
def chargement(nomf):
```

Q24. Écrire une fonction nommée `simulerStrategie` qui prend en entrée :

- `clsStrategie` : une classe représentant une stratégie d'ordonnancement (FCFS, SJF, RR).
- `fp` : la file des processus.
- `duree` : un entier indiquant la durée totale de la simulation.
- `q` : paramètre optionnel qui prend par défaut `None` (quantum).

La fonction calcule et retourne un tuple contenant :

- Le pourcentage de processus achevés.
- Le temps de réponse moyen TRM (voir équation 3).
- Le temps d'attente moyen TAM (voir équation 4).

Ces grandeurs sont mesurées pour une simulation de la stratégie indiquée par la stratégie `clsStrategie` durant `duree` (avec un quantum `q` pour la stratégie RR).

Espace de réponse pour Q24

```
def simulerStrategie(clsStrategie, fp, duree, q = None):
```

Q25. Écrire un script qui permet d'effectuer les étapes suivantes :

- Charger la file des processus à partir du fichier texte "Processus.txt" (supposé existant).
- Demander à l'utilisateur de saisir deux entiers positifs d et q décrivant respectivement la durée totale de la simulation et le quantum à utiliser pour la méthode préemptive.
- Initialiser 3 Matrices `mTermine`, `mTAM` et `mTRM` (`np.ndarray`) ayant 3 lignes et d colonnes où
 - Chaque indice de ligne i représente une stratégie d'ordonnancement (0 pour FCFS, 1 pour SJF et 2 pour RR).
 - Chaque indice de colonne j représente une durée totale de simulation ($j+1$) en cycles d'horloge.
- La simulation se déroule comme suit :
 - Pour chaque stratégie d'ordonnancement i (indice de ligne) et pour chaque indice de colonne j .
 - Copier la liste des processus dans un objet `cfp` (utiliser la fonction `deepcopy` du module `copy` voir Annexe).
 - Remplir `mTermine`, `mTAM` et `mTRM` à partir du résultat de la fonction `simulerStrategie` tel que :
 - `mTermine[i, j]` contiendra le pourcentage de processus terminés suite à la simulation de la stratégie i pour une durée totale de $j+1$.
 - `mTAM[i, j]` contiendra le temps d'attente moyen suite à la simulation de la stratégie i pour une durée totale de $j+1$.
 - `mTRM[i, j]` contiendra le temps de réponse moyen suite à la simulation de la stratégie i pour une durée totale de $j+1$.
- Finalement tracer les courbes illustratives des pourcentage de processus achevés, temps d'attente moyen et temps de réponse moyen pour chaque stratégie et chaque durée de simulation.

Partie III : Base de Données Relationnelle

Le ministère du transport fait appel à votre expertise pour manipuler une base de données relationnelle dans l'objectif de gérer en partie l'infrastructure ferroviaire à l'échelle nationale.

Règles de gestion et contraintes à considérer

- Dans notre contexte une gare ferroviaire de voyageurs est un lieu d'arrêt des trains pour la montée et la descente des passagers.
- Une gare peut jouer le rôle d'arrêt, de départ et/ou arrêt transitoire et/ou arrêt de terminus.
- Dans une gare de départ ou de terminus un train est mis au repos pendant un certain nombre d'heures.
- Une ville peut contenir une ou plusieurs gares.
- Un voyage permet de desservir un ensemble de gares.
- Tout voyage par train doit nécessairement avoir une gare de départ, une gare de terminus et éventuellement un ensemble de gares d'arrêts transitoires.

Le schéma relationnel de la base de données ayant le nom "ferroviaire.db" est défini par les relations suivantes :

■ Ville(idVille, nomVille, nbHabitants)

La table Ville, stocke les différentes informations relatives aux villes, est caractérisée par les attributs suivants :

- idVille : identifiant pour distinguer les différentes villes, clé primaire de type entier.
- nomVille : dénomination d'une ville, de type chaîne de caractères.
- nbHabitants : nombre d'habitants d'une ville, de type entier.

■ Gare(idGare, nomGare, #idVille)

La table Gare, stocke les informations des gares, est caractérisée par les attributs suivants :

- idGare : identifiant de la gare, clé primaire de type chaîne de caractères.
- nomGare : dénomination distinctive attribuée à la gare de type chaîne de caractères.
- idVille : référence le numéro de la ville où se trouve la gare, de type entier, clé étrangère qui fait référence à la table Ville.

■ Train(idTrain, dateMiseService, capacite)

La table Train, stocke les informations relatives aux trains, est caractérisée par :

- idTrain : identifiant du train, clé primaire de type chaîne de caractères.
- dateMiseService : date de mise en service du train, de type Date ("AAAA-MM-JJ")
- capacite : capacité d'accueil maximale du train, de type entier.

■ Voyage(idVoyage, dateHeureDepart, dateHeureArrivee, #idGareDepart, # idGareTerminus, #idTrain)

La table Voyage, stocke les informations des voyages, est caractérisée par les attributs suivants :

- idVoyage : identifiant du voyage, clé primaire de type chaîne de caractères.

- ```
■ Desservir(#idVoyage, #idGareArret, dateHeurePassage)
```

- **idVoyage** : identifiant d'un voyage, de type chaîne de caractères, clé étrangère qui fait référence à la table **Voyage**.
- **idGareArret** : identifiant d'une gare de type chaîne de caractères, clé étrangère qui fait référence à la table **Gare**.
- **dateHeurePassage** : date et heure de passage d'un train à la gare, de type datetime ("AAAA-MM-JJ HH:MM:SS").

Il est aussi à noter que la table `Desservir` ne stocke pas les gares de départ et de terminus pour un voyage donné.

## Algèbre relationnelle

**Q26.** Donner les noms des villes qui contiennent des gares desservies par le voyage d'identifiant "V23".

 Espace de réponse pour Q26

**Q27.** Quels sont les noms des villes dont le nombre d'habitants dépasse 100000 et qui ne contiennent aucune gare.

 Espace de réponse pour Q27

## SQL

**Q28.** Écrire la requête SQL qui permet de créer la table `Desservir` en respectant les contraintes d'intégrité déjà su-citées. Les autres tables sont supposées déjà créées.

 Espace de réponse pour Q28

Dans la suite on suppose que les tables de la base de données sont remplies en respectant les contraintes et les règles de gestion su-citées.

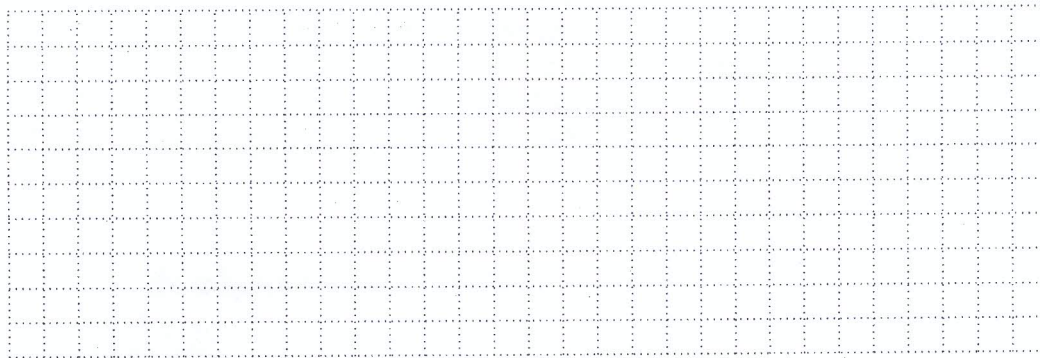
Répondre aux questions suivantes par des requêtes SQL.

**Q29.** Si le train d'identifiant "TR77" circule le premier juin 2023 à 8h, déterminer toutes les informations relatives à son voyage.

### Espace de réponse pour Q29

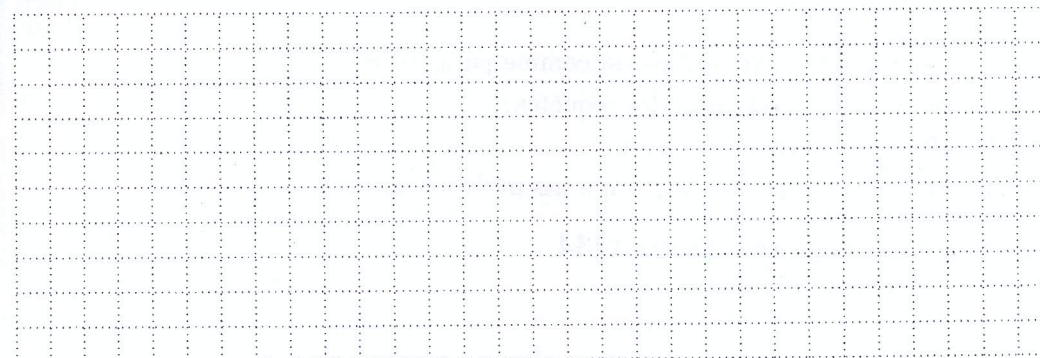
Q30. Déterminer le nombre de gares pour chaque ville.

 Espace de réponse pour Q30



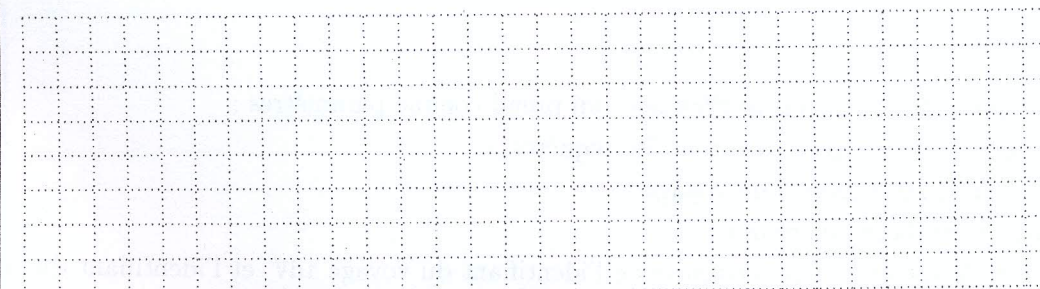
Q31. Donner toutes les informations relatives aux gares qui sont à la fois gare de départ et gare terminus pour un même voyage.

 Espace de réponse pour Q31



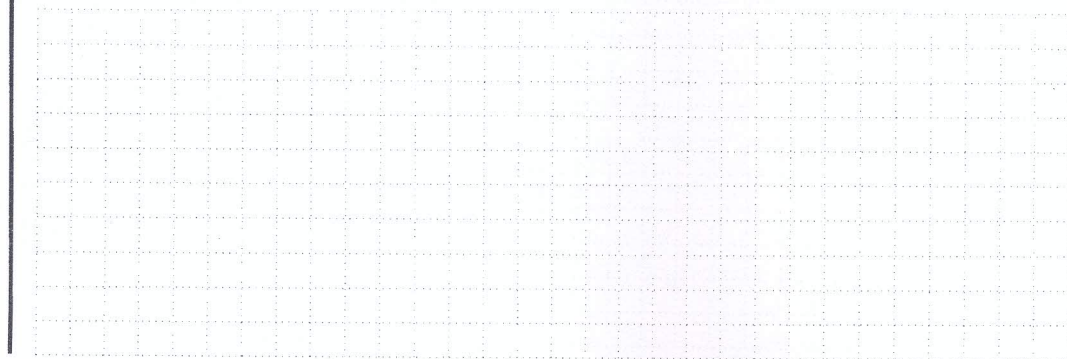
Q32. Déterminez pour chaque voyage le nom de la ville de départ, le nom de la ville d'arrivée ainsi que les dates et heures de départ et d'arrivée.

 Espace de réponse pour Q32



**Q33.** Donner toutes les informations des voyages qui desservent le maximum de gares.

**Espace de réponse pour Q33**



## SQLITE

Dans la suite les fonctions demandées doivent être écrites en Python en désignant par `cur` le curseur d'exécution de requêtes.

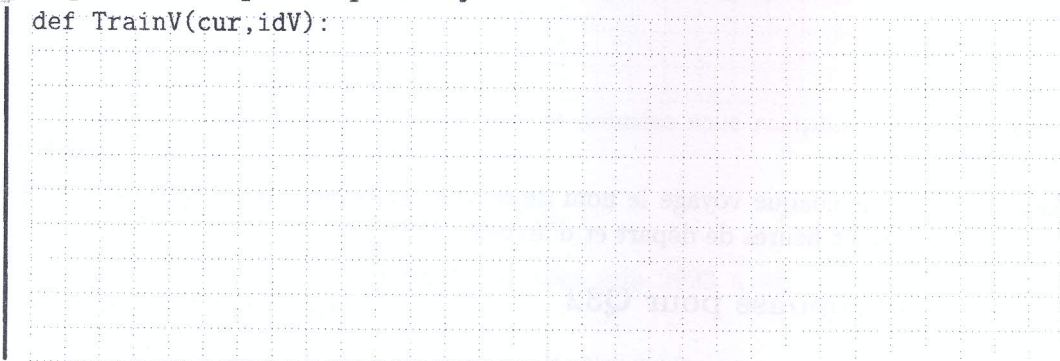
**Q34.** Écrire la fonction `trainV` qui prend comme paramètres :

- `cur` : le curseur d'exécution des requêtes ;
- `idV` : l'identifiant d'un voyage ;

retourne `idT` l'identifiant du train qui a assuré le voyage `idV`.

**Espace de réponse pour Q34**

```
def TrainV(cur, idV):
```



**Q35.** Écrire la fonction `circuitVoyage` qui prend comme paramètres :

- `cur` : le curseur d'exécution des requêtes ;
- `idV` : l'identifiant d'un voyage ;

retourne un dictionnaire dont :

- la clé est un tuple composé de l'identifiant du voyage `idV` et l'identifiant du train qui l'assure.
- la valeur est une liste de tuples où chaque tuple contient le nom de la ville, la gare, la date et heure dans un ordre chronologique croissant. L'extrait qui suit montre un exemple :

```

{('V01', 'TR10'):
 [('GABES', 'GARE-GAB', '2023-01-15 11:15:00'),
 ('SFAX', 'GARE-MAH', '2023-01-15 12:30:00'),
 ('SFAX', 'GARE-SF', '2023-01-15 13:15:00'),
 ('SOUSSE', 'GARE-SOU', '2023-01-15 15:15:00'),
 ('TUNIS', 'GARE-TUN', '2023-01-15 18:00:00')]
}

```

### Espace de réponse pour Q35

def CircuitVoyage(cur,idV):

**Q36.** Écrire la fonction `circuitsVoyages` qui prend en paramètre `cur` et retourne la liste des dictionnaires des circuits de tous les voyages en faisant appel à la fonction `circuitVoyage` de la question **Q35**. L'extrait qui suit montre un exemple :

```
[
 {('V01', 'TR10'):
 [('GABES', 'GARE-GAB', '2023-01-15 11:15:00'),
 ('SFAX', 'GARE-MAH', '2023-01-15 12:30:00'),
 ('SFAX', 'GARE-SF', '2023-01-15 13:15:00'),
 ('SOUSSE', 'GARE-SOU', '2023-01-15 15:15:00'),
 ('TUNIS', 'GARE-TUN', '2023-01-15 18:00:00')
]},
 {('V02', 'TR12'):
 [('TUNIS', 'GARE-TUN', '2023-03-01 16:00:00'),
 ('SOUSSE', 'GARE-KAL', '2023-03-01 17:15:00'),
 ('SOUSSE', 'GARE-SOU', '2023-03-01 18:00:00')
]}
]
```

### Espace de réponse pour Q36

```
def CircuitsVoyages(cur):
```