Semestre 2 – Sujet n°2

Ce sujet est constitué de deux problèmes indépendants. La qualité de la rédaction sera un aspect déterminant dans la notation.

Problème 1. Codes de Gray

Dans tout le problème, n désigne un entier naturel non nul. Si $x \in \{0,1\}$, $\bar{x}$ désigne $1-x$. On note $\oplus$ l'opérateur « ou exclusif » (également appelé XOR), défini par la table suivante.

	0	**1**
0	0	1
1	1	0

On prolonge cette définition à $\mathbb{N}$ de la manière suivante. Soit $(x,y) \in \mathbb{N}^2$ vérifiant $x < 2^p$ et $y < 2^p$ où p est un entier naturel non nul. On décompose x et y en base 2 :

$$x = \sum_{k=0}^{p} a_k 2^k \text{ et } y = \sum_{k=0}^{p} b_k 2^k,$$

où les coefficients a_k et b_k sont des éléments de $\{0,1\}$. On définit alors $x \oplus y$ par

$$x \oplus y = \sum_{k=0}^{p} (a_k \oplus b_k) 2^k.$$

On cherche à obtenir tous les n-uplets composés de 0 et 1 de deux manières différentes.

Attention. On utilisera le type `list` pour représenter des n-uplets (et pas le type `tuple`).

1.1 Ordre lexicographique

Question 1.1. Écrire une fonction `suivant` transformant un n-uplet en son suivant dans l'ordre lexicographique. Par exemple, si `t = [0, 1, 0, 0, 1, 1, 1]`, après exécution de `suivant(t)`, on a `t = [0, 1, 0, 1, 0, 0, 0]`. Cette fonction modifie la liste qu'elle reçoit en argument. De plus elle renvoie un booléen, valant vrai si elle a pu déterminer un n-uplet suivant, ou bien faux si le n-uplet fourni en argument était le dernier. Dans ce dernier cas, la valeur de ce n-uplet après exécution de la fonction est non spécifiée.

Question 1.2. Écrire une fonction affichant tous les n-uplets dans l'ordre lexicographique. *On pourra commencer par écrire une fonction affichant les éléments d'un n-uplet.*

1.2 Ordre de Gray

Pour certaines applications, par exemple pour éviter des états transitoires intermédiaires dans les circuits logiques ou pour faciliter la correction d'erreur dans les transmissions numériques, on souhaite que le passage d'un n-uplet au suivant ne modifie qu'un seul bit. Dans un brevet de 1953, Frank Gray [1] définit un ordre des n-uplets possédant cette propriété.
Pour produire la liste des n-uplets dans l'ordre de Gray, un algorithme consiste à partir de la liste des 1-uplets $(0,1)$ et à construire la liste des $(n+1)$-uplets à partir de celles de n-uplets en ajoutant un 0 en tête de chaque n-uplet puis un 1 en tête de chaque n-uplet en parcourant leur liste à l'envers. On obtient ainsi pour $n = 2$, la liste $(00, 01, 11, 10)$, pour $n = 3$, la liste $(000, 001, 011, 010, 110, 111, 101, 100)$, etc.
On rappelle que chaque n-uplet est une liste, et qu'on travaille donc avec des listes de listes.

1. Frank Gray (1887-1969) était un chercheur travaillant chez Bell Labs ; il a en particulier produit de nombreuses innovations dans le domaine de la télévision.

Question 1.3. Écrire une fonction `ajout` telle que si `a` est un entier et `L` une liste de n-uplets d'entiers, `ajout(a, L)` renvoie une liste contenant les éléments de `L` auxquels on a ajouté `a` en tête.

Question 1.4. Écrire deux fonctions mutuellement récursives `monte` et `descend` prenant en argument un entier n et renvoyant les n-uplets dans l'ordre de Gray. L'une les renverra de $00\cdots0$ à $10\cdots0$, l'autre en sens inverse.

Question 1.5. Évaluer la complexité de ces fonctions en termes d'appels à `monte` et `descend`.

Question 1.6. Décrire une façon simple d'améliorer cette complexité.

1.3 Numérotation des codes de Gray

Dans tout ce qui suit, l'expression représentation binaire, désigne la représentation traditionnelle en base 2. Étant donné $k \in \mathbb{N}^*$, on a de manière unique

$$k = \sum_{i=0}^{p} a_i 2^i \text{ avec } p \in \mathbb{N}, a_i \in \{0,1\}, a_p \neq 0.$$

La représentation binaire canonique de k est $a_p \cdots a_0$ (le bit de poids fort, qui est non nul, en premier). On dira que tout n-uplet constitué d'un nombre quelconque de 0 suivis de la représentation binaire canonique de k est *une* représentation binaire de k.
On définit une fonction g sur $\mathbb{N}$ de la manière suivante. Pour $k \in \mathbb{N}$ et n tel que $k < 2^n$, on considère la liste dans l'ordre de Gray des 2^n n-uplets ; on indice cette liste de 0 à $2^n - 1$; $g(k)$ est le nombre dont une représentation binaire est le n-uplet d'indice k.
Par exemple, si on énumère les 3-uplets selon l'ordre de Gray, on a $(000, 001, 011, 010, 110, 111, 101, 100)$. Dans cette liste, l'élément d'indice 0 est 000 donc $g(0) = 0$, le 3-uplet d'indice 7 est 100 donc $g(7) = 4$.
Soit n un entier naturel et k un entier compris entre 2^n et $2^{n+1} - 1$: $k = 2^n + r$ avec $0 \leq r < 2^n$.

Question 1.7. Démontrer que $g(k) = 2^n + g(2^n - 1 - r)$. *On pourra montrer les représentations binaires de $g(2^n + r)$ et $g(2^n - 1 - r)$ ne diffèrent que par leur premier bit.*

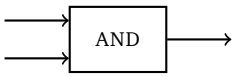
Question 1.8. En déduire que, si la représentation binaire de k est $b_n \cdots b_0$ et si on pose $b_{n+1} = 0$, la représentation binaire de $g(k)$ est $a_n \cdots a_0$ où, pour tout j entre 0 et n, $a_j = b_j \oplus b_{j+1}$. *Raisonner par réucrrence sur k.*

Question 1.9. Exprimer, pour $k \in \mathbb{N}$, $g(k)$ en fonction de k (à l'aide de $\oplus$). *Indication. La multiplication par 2 correspond au décalage des bits d'un cran vers la gauche.*

Question 1.10. On veut savoir calculer l'inverse g^{-1} de g : avec les notations précédentes, exprimer b_j en fonction de a_k , $k \geq j$.

1.4 Implantation matérielle

On cherche à réaliser les fonctions g et g^{-1} avec des circuits logiques. Une porte logique sera représentée par un rectangle contenant son nom (AND, OR, XOR, NOT) et on indiquera les entrées et sorties par des flèches. Par exemple :



On se place d'abord dans le cas $n = 3$.

Question 1.11. Donner un circuit logique à 3 entrées représentant les trois bits d'un entier k inférieur ou égal à 7 et à trois sorties représentant les trois bits de $g(k)$. *On pourra utiliser la porte logique* XOR.

Question 1.12. Donner un circuit pour l'opération inverse.

Question 1.13. Si $n \geq 2$, donner un circuit permettant de passer d'un nombre k à n bits à $g(k)$. Faire de même pour l'opération inverse en utilisant le moins possible de portes. Préciser le nombre de portes utilisées.

1.5 Parcours de sous-ensembles

On souhaite maintenant parcourir toutes les combinaisons de p éléments de l'ensemble $E_n = \{0, \dots, n\}$ avec $0 \leq p \leq n$, c'est-à-dire les sous-ensembles de taille p de E_n. Les éléments d'une combinaison de E_n seront systématiquement considérés dans l'ordre croissant. *Une combinaison est représentée en Python par une liste triée de longueur p.*

Question 1.14. Écrire une fonction d'argument n affichant les combinaisons de 3 éléments pris dans $\{0, \dots, n-1\}$. *On souhaite que ces combinaisons apparaissent dans l'ordre lexicographique. Par exemple pour $n = 4$:* `[0, 1, 2], [0, 1, 3], [0, 2, 3], [1, 2, 3]`.

On revient au cas p entier quelconque entre 1 et n.

Question 1.15. Donner la première et la dernière combinaison de p éléments de E_n ($p \leq n$) lorsqu'on énumère ces combinaisons dans l'ordre lexicographique.

Question 1.16. Écrire une fonction `comb_suivante(c)` permettant de transformer une combinaison en la combinaison suivante dans l'ordre lexicographique. *On suppose que la combinaison en paramètre n'est pas la dernière. On pourra commencer par chercher le plus petit indice j tel que* `c[j+1] > c[j] + 1`.

Question 1.17. En déduire une fonction d'arguments n et p, énumérant et affichant toutes les combinaisons de p éléments de E_n dans l'ordre lexicographique.

Pour améliorer cette énumération, on souhaite passer d'une combinaison à une autre en ne faisant que deux modifications.

Question 1.18. Expliquer comment représenter une combinaison de p éléments pris parmi n par un n-uplet de 0 et 1.

Question 1.19. Modifier les fonctions `monte` et `descend` de la Question 1.4. pour qu'elles renvoient les n-uplets représentant les combinaisons de p éléments pris parmi n, dans l'ordre des codes de Gray.

Question 1.20. Montrer qu'entre deux éléments successifs de la liste de n-uplets renvoyés par monte ou descend, seuls deux bits changent.

Problème 2. Implantation d'une base de données

Introduction

Une requête sur une base de données est décrite au moyen d'un langage *déclaratif*. Le langage SQL est le plus connu. Pour évaluer une requête, un système de gestion de base de données (SGBD) établit un plan d'exécution combinant les opérateurs de l'*algèbre relationnelle*. L'objectif de ce sujet est l'étude de ces opérateurs.
Nous étudierons en première partie l'implantation en Python de ces opérateurs. Nous appliquerons ensuite en deuxième partie ces résultats à des requêtes SQL. Enfin nous verrons en troisième partie omment il est possible de tirer parti des propriétés des données pour améliorer les performances.
Les parties peuvent être traitées indépendamment. Néanmoins, chaque partie utilise des notations et des fonctions introduites dans les parties précédentes.

Bases de données, tables, attributs, enregistrements. Nous détaillons ici la représentation des données dans le modèle relationnel. Une *base de données* est un ensemble de *tables*. Chaque table porte un nom et est associée à un vecteur d'*attributs* de longueur au moins 1. Le nombre d'attributs d'une table est appelé l'*arité* de la table. Le vecteur des attributs $\langle a_0, a_1, \ldots, a_{k-1}\rangle$ d'une table T d'arité k est noté attributs(T) et la table est notée $T[\![a_0, a_1, \ldots, a_{k-1}]\!]$.
Une table $T[\![a_0, a_1, \ldots, a_{k-1}]\!]$ est constituée d'*enregistrements*. La *taille* d'une table est le nombre des enregistrements qu'elle contient. Dans ce sujet, nous considérerons qu'un enregistrement est un vecteur $\langle v_0, v_1, \ldots, v_{k-1}\rangle$ de longueur l'arité k de la table. Chaque élément de ce vecteur est la valeur de cet enregistrement relativement à l'attribut correspondant de la table. La valeur v_i à l'indice i de l'enregistrement est ainsi la valeur associée à l'attribut a_i à l'indice i du vecteur d'attributs de cette table. On pourra donc identifier un attribut et son indice et parler de *la valeur d'un enregistrement associée à un indice*. La valeur d'un enregistrement e associée à un indice i est notée $e[i]$.

Nous considérons dans ce sujet que toutes les valeurs d'attributs sont des chaînes de caractères et que la comparaison entre deux valeurs d'un attribut a un coût unitaire quelles que soient ces valeurs.

Deux enregistrements représentés par des vecteurs contenant les mêmes valeurs aux mêmes indices sont égaux.

Une table peut contenir des enregistrements égaux. L'élimination des enregistrements égaux est une opération complexe qui est l'objet de l'opérateur SQL appelé `DISTINCT`*, que nous étudierons plus loin.*

Exemple (Tables et enregistrements). Considérons une agence de voyages qui vend des trajets et des chambres d'hôtel. La table

$$\texttt{Vehicule}[\![\texttt{IdVehicule, Type, Compagnie}]\!]$$

contient les données relatives aux divers véhicules disponibles. Pour chaque enregistrement e représentant un véhicule dans la table `Vehicule`, la valeur de e associée à l'attribut `IdVehicule` est l'identifiant du véhicule ; la valeur de e associée à l'attribut `Type` est le type de véhicule ; la valeur associée à l'attribut `Compagnie` est le nom de la compagnie qui gère ce véhicule.
Cette table contient trois *enregistrements* qui décrivent des véhicules : un bus de la compagnie *IBUS*, un train de la compagnie *SNCF* et un avion de la compagnie *Hop !*.

$$\langle \texttt{98300}, \texttt{Bus}, \texttt{IBUS}\rangle$$
$$\langle \texttt{1562}, \texttt{TGV}, \texttt{SNCF}\rangle$$
$$\langle \texttt{30990}, \texttt{A320}, \texttt{Hop!}\rangle$$

Considérons d'autre part la table

$$\texttt{Trajet}[\![\texttt{IdTrajet, VilleD, VilleA, DateD, HeureD, IdVehicule}]\!].$$

Cette table contient les trajets élémentaires possibles avec les valeurs des attributs associés : l'identifiant du trajet, la ville de départ, la ville d'arrivée, la date du départ de ce trajet, l'heure de départ du trajet, l'identifiant du véhicule utilisé pour le trajet. On rappelle que toutes ces valeurs sont des chaînes de caractères.
Cette table contient trois trajets possibles le *5 octobre 2016* pour aller de Lille à Rennes. Ils partent respectivement à *9h00*, à *10h00* et à *14h00*.

⟨`Trajet1, Lille, Rennes, 5 oct. 2016, 09h00, 30990`⟩
⟨`Trajet2, Lille, Rennes, 5 oct. 2016, 10h00, 98300`⟩
⟨`Trajet3, Lille, Rennes, 5 oct. 2016, 14h00, 1562`⟩

Représentation des tables et des enregistrements en Python. Dans ce sujet, nous représentons un enregistrement d'une table d'arité k par une *liste Python* de longueur k. L'élément d'indice i de cette liste représente la valeur de l'enregistrement pour l'attribut d'indice i de la table.
Nous représentons une table d'arité k par une liste d'enregistrements. Une table vide est représentée par une liste vide. Voici par exemple une représentation en Python de la table `Vehicule` d'arité 3 :

```
>>> Vehicule
[['98300', 'Bus', 'IBUS'], ['1562', 'TGV', 'SNCF'], ['30990', 'A320', 'Hop !']]
```

Remarque : une table peut être représentée par plusieurs listes différentes. Voici une autre représentation possible de cette table.

```
>>> Vehicule
[['1562', 'TGV', 'SNCF'], ['98300', 'Bus', 'IBUS'],  ['30990', 'A320', 'Hop !']]
```

Complexités des opérations sur les listes Python. Les seules opérations autorisées sur les listes sont les suivantes : `len(L)` (longueur), `L.append(x)` (ajout d'un élément), `L.pop()` (suppression du dernier élément), `L[i]` (accès en lecture ou écriture) et `L1+L2` (concaténation), ainsi que le parcours `for x in L: ...`. Elles ont toutes, sauf le parcours, une complexité $O(1)$. **Il n'est pas autorisé d'utiliser le test d'égalité entre listes dans ce sujet.**

2.1 Implantation des opérateurs de l'algèbre relationnelle en Python

Dans toute la suite, on supposera que les arguments des fonctions Python à rédiger sont bien formés : toutes les listes représentant les enregistrements d'une table ont la même longueur, qui est l'arité de cette table, les entiers représentant des indices d'attributs appartiennent bien à l'intervalle attendu, etc.

Sélection avec test d'égalité à une constante L'opérateur $\sigma_{Constante}$ prend en argument une table T d'enregistrements, un attribut de cette table identifié à son indice i dans le vecteur `attributs`(T) et une valeur c. Il renvoie une table T' associée aux mêmes attributs que T. Elle est constituée des enregistrements de T tels que la valeur de l'attribut d'indice i est égale à la valeur c. Cette table peut être vide.
Exemple. $\sigma_{Constante}$(`Trajet, 1, Lille`) renvoie une table T' avec les mêmes attributs que la table `Trajet`. Elle contient tous les voyages dont la ville de départ est Lille. Dans notre exemple, c'est le cas de tous les voyages. La table T' contient donc les mêmes enregistrements que la table `Trajet`.

Question 2.1. Écrire une fonction `SelectionConstante(table, indice, constante)` qui implante l'opérateur $\sigma_{Constante}$, où `table` est une liste, `indice` un entier et `constante` une chaîne de caractères.

Question 2.2. Donner la complexité de la fonction `SelectionConstante` par rapport à la taille de la table `table`. Justifier votre réponse en vous appuyant sur la structure du programme.

Sélection avec test d'égalité entre deux attributs L'opérateur $\sigma_{Egalite}$ prend en argument une table T d'enregistrements et deux attributs de T identifiés par leurs indices respectifs i et j dans attributs(T) (il est possible que $i = j$), et il renvoie une table T' associée aux mêmes attributs que T. Elle est constituée des enregistrements de T tels que la valeur pour l'attribut d'indice i est égale à la valeur pour l'attribut d'indice j. Cette table peut être vide.
Exemple. $\sigma_{Egalite}$`(Trajet, 1, 2)` renvoie une table avec les mêmes attributs que la table `Trajet`. Elle contient tous les voyages dont la ville de départ est la même que la ville d'arrivée. Le résultat est une table vide.

Question 2.3. Écrire une fonction `SelectionEgalite(table, indice1, indice2)` qui implante $\sigma_{Egalite}$.

Projection sur des indices L'opérateur Π prend en argument une table d'enregistrements $T[\![a_1, \ldots, a_{k-1}]\!]$ d'arité k et un vecteur $L = \langle \ell_0, \ldots, \ell_{k'-1} \rangle$ d'indices identifiant des attributs $\langle a_{\ell_0}, \ldots, a_{\ell_{k'-1}} \rangle$ de la table T, où $0 < k' \leq k$. L'opérateur Π renvoie la table T' d'arité k' associée au vecteur d'attributs $\langle a_{\ell_0}, \ldots, a_{\ell_{k'-1}} \rangle$. Les enregistrement de T' sont obtenus à partir des enregistrements de T en conservant uniquement les valeurs de ces enregistrements pour les attributs de T'. Deux enregistrements distincts et différents de T peuvent ainsi créer deux enregistrements égaux dans T'.

On se restreint au cas où la liste L est ordonnée dans le sens croissant, sans répétition. On supposera que les valeurs du vecteur L sont bien comprises entre 0 et $k-1$.

Exemple. Π(`Trajet`, $\langle 1, 2 \rangle$) renvoie une table associée aux attributs $\langle$`VilleD, VilleA`$\rangle$. Ici, on obtient trois enregistrement égaux à $\langle$`Lille, Rennes`$\rangle$. La table n'en est pas moins constituée de trois enregistrements ; sa taille est 3.

Question 2.4. Écrire une fonction `ProjectionEnregistrement(enregistrement, listeIndices)` qui prend en argument un enregistrement et une liste d'indices $\langle \ell_0, \ldots, \ell_{k'-1} \rangle$ ($0 < k \leq k'$) identifiant des attributs de cette table et renvoie une nouvelle table représentant l'enregistrement d'attributs $\langle a_{\ell_0}, \ldots, a_{\ell_{k'-1}} \rangle$.

Question 2.5. Écrire une fonction `Projection(table, listeIndices)` qui implante l'opérateur Π.

Produit cartésien L'opérateur `X` prend en argument deux tables T_1 et T_2 d'enregistrements. La table T_1, d'arité k_1, est constituée de n_1 enregistrements. La table T_2, d'arité k_2, est constituée de n_2 enregistrements. La table T' résultante est d'arité $k_1 + k_2$, elle possède $n_1 \times n_2$ enregistrements, et son vecteur d'attributs attributs(T') est la concaténation des vecteurs d'attributs de T_1 et de T_2. Les enregistrements de T' sont créés par la concaténation de chaque enregistrement de T_1 avec chaque enregistrement de T_2. Les n_1 premiers attributs sont ceux de T_1 dans l'ordre de T_1, les n_2 suivants sont ceux de T_2, dans l'ordre de T_2. L'ordre des enregistrements ainsi synthétisés dans T' est arbitraire.
Exemple. `X(Vehicule, Trajet)` renvoie une table T'. Les enregistrements de T' sont formés par la concaténation deux à deux des enregistrements de la table `Vehicule` et de ceux de la table `Trajet`.

```
<98300, Bus, IBUS, Trajet1, Lille, Rennes, 5 oct. 2016, 09h00, 30990>
<98300, Bus, IBUS, Trajet2, Lille, Rennes,5 oct. 2016, 10h00, 98300>
<98300, Bus, IBUS, Trajet3, Lille, Rennes, 5 oct. 2016,14h00, 1562>
<1562, TGV, SNCF, Trajet1, Lille, Rennes, 5 oct. 2016, 09h00, 30990>
<1562, TGV, SNCF, Trajet2, Lille, Rennes, 5 oct. 2016, 10h00, 98300>
<1562, TGV, SNCF, Trajet3, Lille, Rennes, 5 oct. 2016,1 4h00, 1562>
<30990 ,A320, Hop!, Trajet1, Lille ,Rennes, 5 oct. 2016, 09h00, 30990>
<30990, A320, Hop!, Trajet2, Lille, Rennes, 5 oct. 2016, 10h00, 98300>
<30990, A320, Hop !, Trajet3, Lille, Rennes, 5 oct. 2016, 14h00, 1562>
```

Question 2.6. Écrire une fonction `ProduitCartesien(table1, table2)` qui implante l'opérateur `X`.

Jointure L'opérateur $\bowtie$ prend en argument deux tables, T_1, d'arité k_1 et de taille n_1, et T_2, d'arité k_2 et de taille n_2. Il prend aussi en argument un attribut de T_1, identifié par son indice i_1 tel que $0 \leq i_1 < k_1$ dans le vecteur attributs(T_1), noté A_1, et un attribut de T_2, identifié par son indice i_2 tel que $0 \leq i_2 < k_2$ dans le vecteur attributs(T_2), noté A_2. Posons $A_2 = \langle a_0, \ldots, a_{i_2}, \ldots, a_{k_2-1} \rangle$.
La table T' résultante est d'arité $k_1 + k_2 - 1$. Son vecteur d'attributs `attribut`(T') est la concaténation du vecteur A_1 et du vecteur $A'_2 = \langle a_0, \ldots, a_{i_2-1}, a_{i_2+1} \ldots, a_{k_2-1} \rangle$, obtenu en effaçant la coordonnée i_2 de A_2. La table T' est constitué d'au plus $n_1 \times n_2$ enregistrements. Les enregistrements de T' sont créés par concaténation des enregistrements e_1 de T_1 et e_2 de T_2 tels que $e_1[i_1] = e_2[i_2]$, en supprimant la valeur d'indice $k_1 + i_2$ pour éviter la répétition avec celle d'indice i_1. L'enregistrement résultant de cette opération est appelé *jointure* des deux enregistrements e_1 et e_2. Il est possible que plusieurs couples (e_1, e_2) produisent des jointures égales dans T'.
Exemple. $\bowtie$ (`Vehicule, Trajet, 0, 5`) renvoie les enregistrements qui décrivent les voyages de chaque véhicule suivi des informations le concernant :

```
<98300, Bus, IBUS, Trajet2, Lille, Rennes, 5 oct. 2016, 10h00>
<1562, TGV, SNCF, Trajet3, Lille, Rennes, 5 oct. 2016, 14h00>
<30990, A320, Hop !, Trajet1, Lille, Rennes, 5 oct. 2016, 09h00>
```

Question 2.7. Écrire une fonction `Jointure(table1, table2, indice1, indice2)` qui implante l'opérateur $\bowtie$. *On pourra commencer par écrire une fonction qui prend en arguments deux enregistrements e_1 et e_2 et deux indices i_1 et i_2 tels que $e_1[i_1] = e_2[i_2]$ et qui renvoie leur jointure au sens ci-dessus.*

Question 2.8. Donner la complexité de `Jointure(table1, table2, indice1, indice2)`. Justifier votre réponse en vous appuyant sur la structure du programme.

Distinct Nous ajoutons aux opérateurs précédents un nouvel opérateur `Distinct` qui n'appartient pas à l'algèbre relationnelle classique. Cet opérateur permet de supprimer les répétitions d'enregistrements égaux dans une table T. Il renvoie une table T' associée aux mêmes attributs que T. Cette table contient exactement un représentant pour chaque classe d'enregistrements égaux de T.
Exemple. `Distinct`(Π(`Trajet`, $\langle 1, 2 \rangle$)) renvoie une table avec un unique représentant de chaque couple possible de villes de départ et d'arrivée. Cette table ne contient qu'un seul enregistrement : $\langle$`Lille, Rennes`$\rangle$.

Question 2.9. Écrire une fonction `SupprimerDoublons(table)` qui implante l'opérateur `Distinct`. *On rappelle que l'opérateur Python d'égalité entre listes ne doit pas être utilisé dans ce sujet. Il est seulement possible de tester l'égalité de deux valeurs associées à un même attribut.*

Question 2.10. Donner la complexité de `SupprimerDoublons(table)`. Justifier votre réponse en vous appuyant sur la structure du programme.

2.2 Implémentation de requêtes SQL en Python

Les données de notre agence de voyage sont enregistrées par les tables suivantes :

- **Vehicule(IdVehicule, Type, Compagnie)** enregistre les véhicules disponibles par l'identifiant du véhicule, son type et sa compagnie.
- **Trajet(IdTrajet, VilleD, VilleA, IdVehicule)** enregistre les trajets élémentaires possibles par l'identifiant du trajet, la ville de départ, la ville d'arrivée ainsi que le véhicule utilisé.
- **Ticket(IdTicket, IdTrajet, Place, Date, Heure, Prix)** enregistre les tickets disponibles par l'identifiant du ticket, celui du trajet auquel ce ticket donne accès, le numéro de la place, la date, l'horaire, le prix.
- **Hotel(IdHotel, Classe, Ville)** enregistre les hôtels connus par l'identifiant de l'hôtel, sa classe, sa ville.
- **Chambre(IdReservation, IdHotel, Date, Prix)** enregistre les chambres d'hôtel qui sont disponibles par l'identifiant de réservation à utiliser, l'identifiant de l'hôtel où se trouve la chambre, la date et le prix.

L'objectif de cette partie est d'étudier l'implémentation de requêtes SQL en combinant les fonctions de l'algèbre relationnelle présentées dans la partie précédente. Tout commentaire expliquant et justifiant la traduction sera apprécié.
Par convention, la liste Python représentant une table aura le même nom que cette table. On représentera un attribut avec sa position. Par exemple, l'attribut `IdTrajet` de la table `Trajet` est représenté par l'entier 0.

Dans chaque cas, le résultat de la requête sera affecté à une variable nommée `resultat`. Par exemple, la requête SQL

```
SELECT Vehicule.Compagnie FROM Vehicule
```

pourra être implantée par `resultat = Projection(Vehicule, [2])`, en supposant que la variable Python `Vehicule` représente la table `Vehicule`. Dans des cas plus complexes, on pourra simplifier l'expression en utilisant des variables auxiliaires pour stocker la valeur de certaines sous-expressions, comme dans l'exemple suivant :

```
r1 = Vehicule
resultat = Projection(r1, [2])
```

*Il est attendu que les candidats rédigent leurs réponses en combinant uniquement les fonctions de l'algèbre relationnelle présentées dans la partie **I**, à l'exclusion de toute autre fonction ou structure de contrôle Python.*

Question 2.11. Proposer une implémentation pour la requête suivante.

```
SELECT *
   FROM Trajet
   WHERE Trajet.VilleD = Rennes
```

Question 2.12. Proposer une implémentation pour la requête suivante.

```
SELECT *
   FROM Trajet, Vehicule
```

Question 2.13. Proposer une implémentation pour la requête suivante.

```
SELECT *
   FROM Trajet, Vehicule
   WHERE Trajet.IdVehicule = Vehicule.IdVehicule
```

Question 2.14. Proposer une implémentation pour la requête suivante.

```
SELECT Classe, Ville, Date, Prix
  FROM Hotel JOIN Chambre
    ON Hotel.IdHotel = Chambre.IdHotel
```

Question 2.15. Proposer une implémentation pour la requête suivante.

```
SELECT Hotel.IdHotel
  FROM Hotel, Trajet, Ticket
    WHERE Hotel.Ville = Trajet.VilleA
      AND  Trajet.IdTrajet = Ticket.IdTrajet
      AND Ticket.Prix = '50'
```

Question 2.16. Proposer une implémentation pour la requête suivante.

```
SELECT *
 FROM Chambre
 WHERE Chambre.Prix = '100'
   AND Chambre.IdHotel IN
     (SELECT Hotel.IdHotel
        FROM Hotel, Trajet, Ticket
         WHERE Hotel.Ville = Trajet.VilleA
           AND  Trajet.IdTrajet = Ticket.IdTrajet
           AND Ticket.Prix = '50')
```

2.3 Amélioration des performances

Il est possible dans certains cas d'améliorer l'implémentation d'une requête en tenant compte de propriétés particulières de la représentation des données ou en utilisant des structures de données supplémentaires. Dans cette partie, nous allons montrer que l'on peut améliorer les performances en triant les données avant de les traiter ou en utilisant des tables associatives (dictionnaires) auxiliaires.

Tables triées par rapport à un indice Une table d'arité k est représentée par une liste d'enregistrements, eux-mêmes représentés par des listes à k éléments. Supposons tout d'abord avoir à disposition une fonction `TrieTableIndice(table, indice)` qui trie par ordre croissant suivant l'ordre lexicographique les enregistrements de la liste `table` d'arité k par rapport à la valeur de l'attribut d'indice `indice` dans le vecteur des attributs de cette table. On suppose que la valeur `indice` est strictement inférieure à k.
Par exemple, la liste `Trajet` ci-dessous est triée par rapport à l'attribut d'indice 1 pour l'ordre lexicographique sur les chaînes de caractères de Python.

```
>>> Trajet
[['30990', 'A320', 'Hop!'], ['98300', 'Bus', 'IBUS'], ['1562', 'TGV', 'SNCF']]
```

Question 2.17. Écrire une fonction `VerifieTrie(table, indice)` qui renvoie `True` si la table est triée pour l'indice `indice` et `False` sinon.

Question 2.18. On considère la fonction `SelectionConstante(table, indice, constante)`. de la Question 2.1.. On suppose que la liste représentant la table `table` est triée selon l'indice `indice`.
Proposer une nouvelle implantation de cette fonction qui utilise cette hypothèse pour en améliorer les performances.

Question 2.19. On considère la fonction `Jointure(table1, table2, indice1, indice2)` de la Question 2.7.. On suppose que les enregistrements de la table `table1` ont des valeurs deux à deux distinctes pour l'attribut d'indice `indice1` et qu'il en va de même pour les enregistrements de la table `table2` avec l'indice `indice2`. On suppose de plus que la liste représentant la table `table1` est triée selon l'indice `indice1` et que celle représentant la table `table2` est triée selon l'indice `indice2`.
Proposer une nouvelle implantation de cette fonction qui utilise ces hypothèses pour en améliorer les performances.

Question 2.20. Donner la complexité de la nouvelle implantation de `Jointure` en vous appuyant sur la structure du programme. Donner des exemples pour lesquels cette nouvelle approche est plus performante. Y a-t-il des cas où elle n'est pas plus performante ?

Utilisation d'un dictionnaire (index) Les opérations suivantes sur les dictionnaires ont une complexité $O(1)$: `dico = {}` (initialisation), `dico[c]` (accès en lecture ou écriture), `c in dico` (test d'appartenance). On rappelle qu'on peut itérer sur les clés présentes dans un dictionnaire à l'aide de `for c in dico: ...`.

Les réponses doivent être exclusivement rédigées à l'aide de ces opérations sur les dictionnaires.

Voici un exemple d'utilisation :

```
>>> dico = {}
>>> dico['aaa'] = [1]
>>> dico['bbb'] = [2]
>>> dico['ccc'] = [3]
>>> dico['aaa'].append(4)
>>> dico['ccc'] = [5]
>>> for c in dico: print(c, ' -> ', dico[c])
aaa -> [1,4]
bbb -> [2]
ccc -> [5]
>>> dico['ddd']
KeyError: 'ddd'
```

Utilisation de dictionnaires pour indexer les bases de données Considérons une table T d'arité k et de taille n représentée par une liste Python d'enregistrements $L = [e_0, \ldots, e_{n-1}]$. Considérons un indice i d'attribut de T tel que $0 \leq i < k$. On peut associer à T et i un dictionnaire Python $\texttt{Dico}_{T,i}$ de la manière suivante. Les clés de ce dictionnaire sont les valeurs possibles v qui apparaissent pour l'attribut d'indice i dans la table T. L'image associée à une clé v est la liste des positions dans L des enregistrements e tels que $e[i] = v$. Si l'attribut d'indice i de T ne prend la valeur v pour aucun enregistrement, cette clé n'est pas enregistrée dans le dictionnaire. L'image d'une clé est donc une liste non vide.
Exemple (Dictionnaire associé à une table). Considérons la table

$$\texttt{Vehicule}[\![\texttt{IdVehicule}, \texttt{Type}, \texttt{Compagnie}]\!]i$$

avec les enregistrements suivants :

```
<98300, Bus, IBUS>
<1562, TGV, SNCF>
<30990, A320, Hop !>
<1789, TGV, SNCF>.
```

Soit `dico` le dictionnaire $\texttt{Dico}_{\texttt{Vehicule},1}$ associé à l'attribut `Type` de position 1. Nous avons

```
>>> for c in dico: print(c, ' -> ', dico[c])
Bus -> [0]
A320 -> [2]
TGV -> [1, 3]
>>> dico['Ariane6']
KeyError: 'Ariane6'
```

Application à la sélection

Question 2.21. Écrire une fonction `CreerDictionnaire(table, indice)` qui prend en argument une table `table` et un indice `indice` d'attribut de `table`, et qui renvoie un dictionnaire de la table `table` pour l'attribut d'indice `indice`.

Question 2.22. Écrire une fonction `SelectionConstanteDictionnaire(table, indice, constante, dico)` qui a la même fonctionnalité que `SelectionConstante` de la Question 2.1., mais qui prend en plus en argument un dictionnaire `dico` de la table `table` pour l'indice `indice`.

Question 2.23. Comparer la complexité de la fonction `SelectionConstanteDictionnaire` avec celle de la fonction `SelectionConstante`. Donner des exemples pour lesquels cette nouvelle approche est plus performante. Y-a-t-il des cas où elle n'est pas plus performante ?

Application à la jointure

Question 2.24. Écrire une fonction `JointureDictionnaire(table1, table2, indice1, indice2, dico2)` qui a la même fonctionnalité que `Jointure` de la Question 2.7., mais qui prend en plus en argument un dictionnaire `dico2` de la table `table2` par rapport à l'indice `indice2`.

Question 2.25. Donner la complexité de cette fonction par rapport aux tailles et arités respectives des tables `table1` et `table2` ainsi que par rapport à la longueur maximale d'une liste renvoyée par le dictionnaire `dico2`, qui sera notée k_2. Justifier votre réponse en vous appuyant sur la structure du programme.

Question 2.26. L'opérateur de jointure prend en argument deux tables qui jouent des rôles analogues. Il serait donc possible d'utiliser un dictionnaire pour `table1` au lieu d'un dictionnaire pour `table2`. Comment pourrait-on choisir la table à indexer pour obtenir les meilleures performances ?