Polynésie - mars 2021 (corrigé)

Exercice 1 (Programmation et tris)
Partie A : manipulation d'une liste en python.

1. Après exécution, il sera affiché 8 (le nombre d'éléments dans la liste `notes`), puis la liste modifiée soit `[8,7,18,16,12,9,17,3]`.
2. Les deux codes suivants conviennent :

```
print(notes[2:5])
```

```
for i in range(2,5) :
    print(notes[i])
```

Partie B : tri par insertion.

1. Le code suivant convient :

```
def tri_insertion(liste : list) -> None :
    for indice_courant in range(1, len(liste)) :
        element_a_inserer = liste[indice_courant
        i = indice_courant - 1
        while i >= 0 and liste[i] > element_a_inserer :
            liste[ i+1 ] = liste[ i ]
            i = i - 1
        liste[ i+1 ] = element_a_inserer
```

2. Après le 1er passage de la boucle `for`, nous avons trié tous les éléments jusqu'à l'indice 1 compris, ainsi `notes = [7,8,18,14,12,9,17,3]`
3. Après le 3ème passage de la boucle `for`, nous avons trié tous les éléments jusqu'à l'indice 3 compris, ainsi `notes = [7,8,14,18,12,9,17,3]`

Partie C : tri fusion.

1. Le tri fusion est un algorithme récursif, puisque on rappelle la même fonction sur deux sous-tableaux du tableau initial.
2. Afin d'assembler deux tas déjà triés, on parcours chacun des tas à partir des deux plus petites cartes. Tant qu'il reste des cartes visibles, on retire la plus petite carte visible et on la met dans le nouveau tas trié. Au bout du compte, un des deux tas sera vidé, on termine alors avec toutes les cartes du tas restant.
3. Le code suivant convient :

```
def tri_fusion(liste : list, i_debut : int, i_fin : int) -> None :
    if i_debut < i_fin :
        i_partage = floor((i_debut + i_fin) / 2) # <=> ( i_debut + i_fin ) // 2
        tri_fusion(liste, i_debut, i_partage)
        tri_fusion(liste, i_partage+1, i_fin)
        fusionner(liste, i_debut, i_partage, i_fin)
```

4. La ligne `from math import floor` permet de charger la définition de la fonction `floor()` définie dans la bibliothèque `math`.

Partie D : comparaison du tri par insertion et du tri fusion.

1. L'algorithme utilisé a été un tri fusion, puisque chaque sous-tableau est assemblé avec son voisin afin de formé un tableau deux fois plus grand.

2-3. Le tri par insertion possède une complexité en $O(n^2)$ puisqu'il y a deux boucles imbriquées : n tours de boucles principales qui requièrent $n - i$ tours de boucles.
Le tri fusion possède une complexité en $O(n \log_2(n))$ puisqu'il y a $\log_2(n)$ divisions de tableau avec au pire n itérations à chaque fois.

Exercice 2 (SQL)

Partie A : modèle relationnel.

1. La clef primaire de la table `Clients` est l'entier `IdClient`, tandis que celle de la table `Articles` est l'entier `IdArticle`.
2. L'attribut `Email` est une chaîne de caractères possèdant au maximum 50 caractères, tandis que l'attribut `Quantite` est un entier.
3. Le code suivant convient :

```
CREATE TABLE Commandes (
    IdCmd INT PRIMARY KEY,
    IdClient INT,
    Date DATE,
    AdresseLivraison VARCHAR(90),
    PaiementValide BOOLEAN,
    LivraisonFaite BOOLEAN,
    FOREIGN KEY(IdClient) REFERENCES Clients(IdClient)
);
```

Partie B : site web.

1. Contrairement à la méthode `GET`, la méthode `POST` ne rend pas visible dans l'url les réponses entrées dans le formulaire par l'utilisateur, cette dernière est donc préférable. De plus, la méthode `GET` est limitée en taille de caractères.
2. Afin de chiffrer un paiement et cacher les informations sensibles, il vaut mieux utiliser le protocole HTTPS qui est sécurisé.
3. Il est utile de vérifier le format des informations saisies dans un formulaire afin de ne pas avoir d'injection de code indésirable.

Partie C : requêtes SQL.

1. La requête suivante convient :

```
SELECT IdArticles, Libelle FROM Article
WHERE PrixEnCentimes <= 1500
```

2. La requête permet de récupérer les identifiants clients, leur courriel, l'identifiant ainsi que l'adresse de livraison des clients qui n'ont pas eu leur paiement validé.
3. On effectue une jointure entre les tables `Articles` et `ArticlesCommande` :

```
SELECT a.Libelle FROM Articles AS a
INNER JOIN ArticlesCommande AS ac ON ac.IdArticle = a.IdArticle
WHERE ac.IdCmd = 1345
```

4. La requête suivante convient :

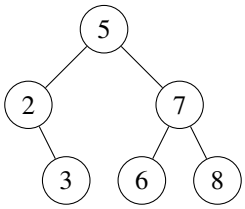
```
INSERT INTO Articles (Libelle, Description, PrixEnCentimes)
VALUES ("imperméable", "Cet imperméable se replie en forme de pochette", 999)
```

Partie D : adaptation du modèle relationnel.

1. Afin de comptabilisé le stock des articles il faudrait ajouter un attribut `stock` de type entier dans la table `Articles`.
   Afin d'avoir une adresse de livraison par défaut, il faudrait ajouter une table `Adresse` qui contiendrait l'adresse proprement dite et qui soit reliée à la table `Clients`. Il faut donc ajouter dans la table `Clients` l'attribut `IdAdresse`. L'adresse de livraison sera alors celle du client (si absente) ou bien celle spécifiée.
2. Cet algorithme pose problème lors de la gestion du stock. Il faut remplacer `Stock - 1` par `Stock - Quantité`.

Exercice 3 (Arbre binaire de recherche et POO)
Partie A : étude d'un exemple.

1. La racine de cet arbre binaire possède la valeur 5 et deux fils de valeurs 2 et 7.
2. Les nœuds de la branche se terminant par 3 ont pour valeurs 5, 2 et 3.
3. L'arbre binaire de recherche résultant de l'ajout de la valeur 6 sera comme suit :



Partie B : implémentation en Python.

1. La méthode `__init__()` se nomme le constructeur et est appelée lorsque le programmeur créé un nouvel objet de type `ABR`.
2. Dans cette implémentation, si la valeur existe (`e==self.valeur`), l'élément n'est pas ajouté. Il ne peut y avoir de nœud ayant deux valeurs identiques.
3. Le code suivant convient :

```
arbre = ABR(5)
arbre.insererElement(2)
arbre.insererElement(3)
arbre.insererElement(7)
arbre.insererElement(8)
```

Partie C : tri par arbre binaire de recherche.

1. Si l'on parcourt un arbre binaire de recherche avec un parcours en profondeur de manière pré-fixé, la valeur des nœuds visités sera croissante.
2. Les tris par insertion ou sélection ont une complexité temporelle quadratique ($O(n^2)$) tandis que le tri dans un arbre binaire de recherche est en moyenne quasi-linéaire en $O(n\log_2(n))$. Néanmoins si les valeurs sont déjà ordonnées, l'arbre binaire sera un arbre filaire (similaire à une liste) et le tri sera quadratique ($O(n^2)$).

Exercice 4 (Architecture matérielle, SOC, réseaux et système d'exploitation)
Partie A : routage dans un réseau informatique.

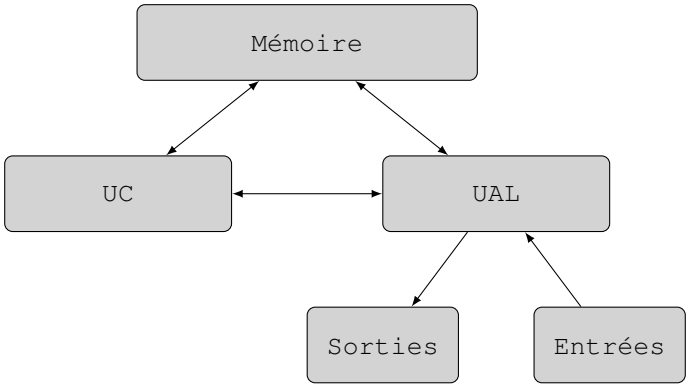
1. Le protocole TCP/IP prévoit un découpage en paquets afin que tous les paquets aient une taille similaire, ce qui permet de fluidifier les communications : chacun des paquets ne suivant pas la même route et en cas de perte, seuls les paquets perdus doivent être renvoyés. Le protocole prévoit une encapsulation des données afin que la couche réseau connaisse toutes les informations nécessaires à cette couche pour faire transiter les données sans se soucier des autres étapes.
2. Les sommets du graphe seront les machines ou les routeurs tandis que les arêtes seront les voies de communication des informations (câble RJ45, fibre, wifi, etc.) Le protocole RIP utilise le nombre de sauts comme poids des arêtes.

Partie B : système d'exploitation.

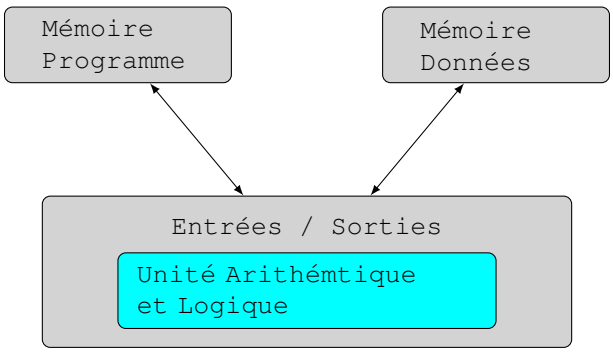
1. Une situation d'interblocage consiste en un blocage des processus qui ont besoin des mêmes ressources pour fonctionner.
   Par exemple, s'il existe deux processus A et B qui chacun nécessite deux ressources 1 et 2 afin de terminer leur action.
   Le processus A peut bloquer la ressource 1 en vue de son utilisation future, tandis que le processus B bloque la ressource 2. Pour se terminer et libérer la ressource 1, le processus A doit avoir accès à la ressource n2 (qui est bloquée par B), mais il en va de même pour le processus B qui attend l'accès à la ressource 1 avant de se terminer et libérer la ressource 2. Les processus A et B s'attendent donc réciproquement, il y a interblocage.
2. Afin d'éviter tout interblocage, il faut retirer l'une des conditions de Coffman (exclusion mutuelle, ressource holding, réquisition impossible, attente circulaire). Ainsi on peut allouer toutes les ressources nécessaires à un processus avant de le lancer, ou bien ignorer l'exclusion mutuelle.

Partie C : architecture matérielles.

1. On a le schéma suivant :



2. Le compteur de programme (Instruction Pointer) se trouve dans la partie `Unit Control`, et sert à indiquer l'adresse mémoire contenant l'instruction en cours de traitement.
   Le registre d'instruction (Instruction Register) est également dans la partie `Unit Control`, et sert à contenir l'instruction en cours de traitement.
3. On a le schéma suivant :



4. Une mémoire morte est une mémoire qui ne s'efface pas lorsque l'alimentation électrique s'arrête et dont le contenu est fixé, non modifiable. Une mémoire vive est une mémoire qui peut être réécrite et souvent rapide d'accès.
   Puisque les microcontrôleurs ne sont pas évolutifs et sont affectés à une tâche particulière, leur programme n'a pas besoin d'être changé : l'écrire dans une mémoire ROM suffit donc.

Partie D : système sur puce.

1. Avoir plusieurs processeurs permet de paralléliser les tâches.
2. Les systèmes sur puce peuvent intégrer des bus ayant des tailles ou des vitesses différentes selon l'utilisation qui en sera faite, les bus sont donc créés sur-mesure pour répondre aux besoins du système. Néanmoins, une certaine standardisation des bus commence à apparaître.
3. Un circuit imprimé de petite taille consomme moins d'énergie, et permet à taille égale d'avoir une plus grande puissance de calcul.
4. La miniaturisation des circuits imprimés se heurte au phénomène de surchauffe (la chaleur ayant du mal se dissiper suffisamment) ainsi qu'à la limite des phénomènes électriques à l'échelle atomique.

Exercice 5 (Données en table et SQL)
Partie A : modèle relationnel.

1. La relation `Film` possède l'attribut `IdFilm` comme clef primaire, tandis que la relation `Abonnes` possède l'attribut `IdAbonne`.
2. L'attribut `IdFilm` est de type entier, tandis que l'attribut `Description` est une chaîne de caractères d'au plus 150 caractères.
3. La relation `ComptesAbonnes` possède pour clef primaire l'attribut `IdCpt` et une clef étrangère `IdAbonne` provenant de la relation `Abonnes`.
4. Afin de stocker les acteurs principaux de chaque film, on peut les ajouter à la relation `Films`, ou mieux puisqu'un acteur peut joueur dans plusieurs films de la base de données, créer une relation `Acteurs` qui sera relié à la relation `Films` par sa clef primaire.
5. Afin d'associer une tranche d'âge à un compte abonné, on peut ajouter l'année de naissance de l'abonné dans l arelation `Abonnes` et calculer la tranche d'âge automatiquement. Néanmoins cette solution empêche l'abonné d'avoir plusieurs comptes de différentes tranches (par exemple, des parents qui veulent avoir trois comptes). Ainsi on préfèrera ajouter à la relation `ComptesAbonnes` un attribut `TrancheAge` qui soit un entier (12, 15, 18, 255) tandis que l'on rajoutera à la relation `Films` l'attribut `TrancheAge` qui sera également un entier.

Partie B : requêtes SQL.

1. La requête suivante convient :

```
SELECT IdCpt, Pseudo FROM ComptesAbonnes
WHERE IdAbonne = 237
```

2. La requête effectue la moyenne des étoiles (`AVG(NbEtoiles)`) du film identifié par 1 542.
3. La requête effuctue une jointure (`[INNER] JOIN`) entre les tables `Films` et `Votes` sur l'identifiant `IdFilm` (`ON u.IdFilm = v.IdFilm`). Ainsi elle renvoie tous les identifiants, titres et nombre d'étoiles des votes que le compte abonné n° 508 a fait. L'affichage se fait selon l'ordre décroissant (`ORDER BY v.NbEtoiles DESC`) du nombre d'étoiles.
4. Il s'agit de faire une mise à jour de la table `ComptesAbonnes`. La requête suivante convient donc :

```
UPDATE ComptesAbonnes
SET Pseudo = 'Champion'
WHERE IdCpt = 508
```

Partie C : conseils de films.

1. La fonction `distance()` vérife que la variable `listeFilms` est bien du type `list` non-vide.
   Il renvoie ensuite la moyenne des écarts (en valeur absolue) des votes entre les deux identifiants de comptes fournies en paramètres.
2. En général, le nombre d'étoiles données à un film est un entier compris entre 0 et 5, la moyenne ne pouvant être supérieure à l'écart maximal, il est surprenant de vérifier qu'elle soit inférieure à 10. Néanmoins, on propose le code suivant :

```
def conseilsFilms(IdCpt : int) -> list :
    conseils = []
    meilleursFilms = podiumCompte( IdCpt )
    spectaComptes = spectateurs( meilleursFilms )
    for spCpt in spectaComptes :
        if distance( IdCpt, spCpt, meilleursFilms ) <= 10 :
            spCptPodium = podiumCompte( spCpt )
            conseils += spCptPodium[:3] # on prend les 3 meilleurs
    return conseils
```