Amérique du nord - mai 2024 - sujet 2 (corrigé)

Exercice 1 (Programmation Python et algorithmes de tri)

1. La fonction suivante convient :

```
def echange(tab, i, j):
    temp = tab[i]
    tab[i] = tab[j]
    tab[j] = temp
```

2. Le code suivant convient :

```
def triStooge(tab, i, j):
    if tab[i] > tab[j]:
        echange(tab, i, j)
    if (j - i) > 1:
        k = (j - i + 1)//3
        triStooge(tab, i, j-k)
        triStooge(tab, i+k, j)
        triStooge(tab, i, j-k)
```

3. Puisque la fonction `triStooge` s'appelle elle-même, on a affaire à un algorithme récursif.
4. Entre `i = 0` et `j = 5`, il y a 6 éléments donc `k = (5 - 0 + 1)//3 = 6//3 = 2`.
5. En comptant le nombre de fois où `triStooge` apparaît dans l'arbre d'appels de la figure 1 (sans le premier), on en déduit qu'il y a 39 appels récursifs effectués (ne pas oublier les cases marquées `1`, `2` et `3`.
6. - Case `1` : `triStooge(A, 1, 3)`
   - Case `2` : `triStooge(A, 2, 3)`
   - Case `3` : `triStooge(A, 0, 3)`
7. Puisque $\frac{8}{3} > 2$, le tri par sélection, le tri par insertion et le tri fusion ont un coût strictement meilleur/

Exercice 2 (Bases de données et SQL)

1. On obtient le tableau suivant :

Dufour	Marc
Martin	Sophie

2. La requête suivante convient :

```
SELECT nom_medic FROM medicament WHERE prix < 3
```

3. La requête suivante convient :

```
INSERT INTO client VALUES (3, 'Durand', 'Nathalie', '269054958815780')
```

4. Les attributs qui doivent être déclarés comme clés étrangères sont les suivants :
   - `id_client` qui permet de faire la relation avec la table `client` ;
   - `id_medic` qui permet de faire la relation avec la table `medicament`.
5.
   - Il faut au maximum 6 comprimés de Paracétamol 1 gramme CP, donc il faut 1 boîte pour la ligne 7.
   - Il faut 28 comprimés d'acide ascorbique, donc il faut 3 boîtes pour la ligne 8.
6. La requête suivante convient :

```
UPDATE medicament SET quantite = 447 WHERE id_medic = 4
```

7. Une boîte de Paracétamol 1 gramme CP coûtant $3{,}50$ euros et une boîte d'acide ascorbique coûtant $5{,}50$ euros, on déduit de la question 5 que le coût total des médicaments de Madame Martin est de $1 \times 3{,}50 + 3 \times 5{,}50 = 20$ euros.
8. La requête suivante convient :

```
SELECT nom_medic FROM medicament
JOIN ordonnance ON ordonnance.id_medic = meidcament.id_medic
WHERE id_ordo = 6
```

Exercice 3 (POO, listes chaînées, réseaux, architecture matérielle)
Partie A.

1. L'adresse `192.168.1.3` convient (en fait, le dernier octet peut prendre n'importe quelle valeur entre 3 et 254).
2. On a la liste suivante :

```
[Transaction("Alice", "Charlie", 10), Transaction("Bob", "Alice", 5)]
```

3. Le paramètre `bloc_precedent` du `bloc0` est à `None` car il n'y a pas de bloc précédent (`bloc0` est le premier bloc créé).
4. L'attribut `bloc_precedent` du `bloc1` doit être égal à `bloc0`.
5. Le code Python suivant convient :

```
ma_blockchain = Blockchain()
bloc1 = Bloc([Transaction('Alice', 'Charlie', 50), Transaction('Charlie', 'Bob', 30)],
             ma_blockchain.tete)
ma_blockchain.tete = bloc1
bloc2 = Bloc([Transaction('Bob', 'Charlie', 20), Transaction('Bob', 'Charlie', 20),
             Transaction('Charlie', 'Alice', 30)], ma_blockchain.tete)
ma_blockchain.tete = bloc2
```

6. A l'issue du `bloc2`, le solde de Bob est de $100 + 30 - 20 - 20 = 90$ `nsicoin`.
7. La fonction suivante convient :

```
def ajouter_bloc(self, liste_transactions):
    bloc = Bloc(liste_transactions, self.tete)
    self.tete = bloc
```

8. Il faut utiliser l'adresse de broadcast (adresse de diffusion) : `192.168.1.255`
9. Le code suivant convient :

```
def calculer_solde(self, utilisateur):
    if self.bloc_precedent is None:
        solde = 0
    else:
        solde = self.bloc_precedent.calculer_solde(utilisateur)
        for transaction in self.liste_transactions:
            if transaction.expediteur == utilisateur:
                solde = solde - transaction.montant
            elif transaction.destinataire == utilisateur :
                solde = solde + transaction.montant
        return solde
```

10. Si on a toujours une instance `ma_blockchain` de la classe `Blockchain` :

```
ma_blockchain.tete.calculer_solde('Alice')
```

Partie B.

11. On fait une recherche exhaustive quand on teste toutes les possibilités.
12. Le `bloc0` est le premier bloc, l'attribut `bloc_precedent` du `bloc0` est `None`. L'analyse des lignes 11 à 15 de la classe `Bloc` nous permettent d'affirmer que l'attribut `hash_bloc_precedent` est `'0'`.
13. Le hash étant codé sur 256 bits, il y a 2^{256} possibilités.
14. Le code suivant convient (en partant du principe que la méthode `calculer_hash` utilise `self.nonce` pour effectuer le calcul du hash, ce qui n'est indiqué nulle part !) :

```
def minage_bloc(self):
    self.nonce = 0
    self.hash = self.calculer_hash()
    while self.hash[0] != '0' or self.hash[1] != '0' :
        self.nonce = self.nonce + 1
        self.hash = self.calculer_hash()
```