# Sujet A

❑ **Exercice** : *type A*

On considère le schéma de base de données suivant, qui décrit un ensemble de fabricants de matériel informatique, les matériels qu'ils vendent, leurs clients et ce qu'achètent leurs clients. Les attributs des clés primaires des six premières relations sont soulignés.

— `Production(NomFabricant, Modele)`

— `Ordinateur(Modele, Frequence, Ram, Dd, Prix)`

— `Portable(Modele, Frequence, Ram, Dd, Ecran, Prix)`

— `Imprimante(Modele, Couleur, Type, Prix)`

— `Fabricant(Nom, Adresse, NomPatron)`

— `Client(Num, Nom, Prenom)`

— `Achat(NumClient, NomFabricant, Modele, Quantite)`

Chaque client possède un numéro unique connu de tous les fabricants. La relation `Production` donne pour chaque fabricant l'ensemble des modèles fabriquées par ce fabricant. Deux fabricants différents peuvent proposer le même matériel. La relation `Ordinateur` donne pour chaque modèle d'ordinateur la vitesse du processeur (en Hz), les tailles de la Ram et du disque dur (en GO) et le prix de l'ordinateur (en €). La relation `Portable`, en plus des attributs précédents, comporte la taille de l'écran (en pouces). La relation `Imprimante` indique pour chaque modèle d'imprimante si elle imprime en couleur (vrai/faux), le type d'impression (laser ou jet d'encre) et le prix (en €). La relation `Fabricant` stocke les noms et adresses de chaque fabricant, ainsi que le nom de son patron. La relation `Client` stocke les noms et prénoms de tous les clients de tous les fabricants. Enfin, la relation `Achat` regroupe les quadruplets (client $c$, fabricant $f$, modèle $m$ et quantité $q$) tel que le client de numéro $c$ a acheté $q$ fois le modèle $m$ au fabricant de nom $f$. On suppose que l'attribut `Quantite` est toujours strictement positif.

1. Proposer une clé primaire pour la relation `Achat`, quelles sont les conséquences en terme de modélisation ?

   > On propose (`NumClient`, (`NomFabricant`, `Modele`)) comme clé primaire de la relation `Achat`, `NumClient` est une clé étrangère associée à la clé primaire `Num` de la relation `Client`, `(NomFabricant,Modele)` est une clé étrangère associée à la clé primaire (`NomFabricant`, `Modele`) de la relation `Production`.
   > Lors de l'ajout d'un nouvel enregistrement dans la table `Achat`, on doit vérifier que le client n'a pas déjà acheté ce modèle au même fabricant auquel cas, on mettra à jour le champ quantité afin de préserver l'unicité de la clé primaire.

2. Identifier l'ensemble des clés étrangères éventuelles de chaque table.

   > Dans la relation `Production`, `NomFabricant` est une clé étrangère qui fait référence à la clé primaire `Nom` de la relation `Fabricant`

3. Donner en SQL des requêtes répondant aux questions suivantes :
   a) Quels sont les numéros de modèles des matériels (ordinateur, portable ou imprimante) fabriqués par l'entreprise de nom Durand ?

   ```sql
   SELECT Modele
   FROM Production
   WHERE NomFabricant = "Durand";
   ```

   b) Quels sont les noms et adresses des fabricants produisant des portables dont le disque dur a une capacité d'au moins 500 Go ?

   ```sql
   SELECT Nom, Adresse
   FROM Fabricant
   JOIN Production ON NomFabricant = Nom
   JOIN Portable ON Production.Modele = Ordinateur.Modele
   WHERE Portable.Dd >= 500;
   ```

c) Quels sont les noms des fabricants qui produisent au moins 10 modèles différents d'imprimantes ?

```
SELECT NomFabricant, COUNT(*) AS nb
FROM Production
JOIN Imprimante ON Imprimante.Modele = Ordinateur.modele
GROUP BY NomFabricant HAVING nb>10;
```

d) Quels sont les numéros des clients n'ayant acheté aucune imprimante ?

```
SELECT NumClient FROM Client
EXCEPT
SELECT NumClient From Achat
JOIN Imprimante ON Imprimante.Modele = Achat.Modele
```

❑ **Exercice** : *type B*

Les fonctions demandées dans cet exercice doivent être écrites en langage C. Un fichier contenant le code compagnon de cet exercice est à télécharger à l'adresse `https://fabricenativel.github.io/cpge-info/oraux/`
On dit qu'un tableau `tab` de taille $n$ est *autoréférent* si pour tout entier `i` $\in [\![0; n-1]\!]$, `tab[i]` est le nombre d'occurences de `i` dans `tab`. Par exemple, le tableau `ex={ 1, 2, 1, 0 }` est autoréférent, en effet :

- `ex[0] = 1` et `0` apparaît bien une fois dans le tableau
- `ex[1] = 2` et `1` apparaît bien deux fois dans le tableau
- `ex[2] = 1` et `2` apparaît bien une fois dans le tableau
- `ex[3] = 0` et `3` n'apparaît pas dans le tableau

L'exercice traite de la recherche par retour sur trace d'un tableau autoréférent de taille donnée.

1. Justifier rapidement que dans un tableau autoréférent de taille $n$, chaque valeur doit être comprise entre 0 et $n$ et que la somme des éléments du tableau doit être égale à $n$.

   Si on considère un tableau autoréférent de taille $n$, alors on a $t[i]$ = nombre d'occurrences de $i$ dans $t$, comme une valeur peut apparaître au maximum $n$ fois, on a $t[i] \leq n$. De plus, $t$ de taille $n$, il y a en tout $n$ occurrences de valeurs dans $t$, c'est à dire que la somme des $t[i]$ vaut $n$.

2. Montrer que pour $n \in [\![1; 3]\!]$, il n'existe aucun tableau auto référent de taille $n$.

   Il suffit de tester les possibilités, en utilisant la question précédente.
   - taille 1 : le seul tableau possible est $\{1\}$ et il n'est pas autoréférent.
   - taille 2 : Comme $t[0]$ ne peut pas valoir 0, le seul tableau possible est $\{1, 1\}$ et il n'est pas autoréférent.
   - taille 3 : $t[0]$ ne peut pas valoir 0. Si $t[0] = 1$, alors soit $t[1] = 0$ et donc $t[2] = 2$ soit $t[2] = 0$ et $t[1] = 2$ or aucun de ces tableaux n'est autoréférent. Si $t[0] = 2$ alors $t[1] = 0$ et $t[2] = 0$ et ce ne tableau n'est pas autoréférent.

3. Déterminer un autre tableau autoréférent de taille 4 que celui donné en exemple.

   - si $t[0] = 1$ alors 0 apparait et une seule fois, 1 apparait au moins une fois et la somme faisant 4, les possibilités sont $\{1, 1, 0, 2\}$, $\{1, 1, 2, 0\}$, $\{1, 2, 0, 1\}$ et $\{1, 2, 1, 0\}$. Seul le dernier tableau est autoréférent et c'est celui donné dans l'énoncé.
   - si $t[0] = 2$ alors 0 apparait deux fois, donc les possibilités sont $\{2, 2, 0, 0\}$ , $\{2, 0, 2, 0\}$ et $\{2, 0, 0, 2\}$. Seul le tableau $\{2, 0, 2, 0\}$ est autoréférent.
   - si $t[0] = 3$ alors 0 apparait trois fois, impossible.

   En conclusion, il n'y a qu'une seule autre possibilité que celle donnée dans l'enoncé, c'est le tableau $\{2, 0, 2, 0\}$.

4. Soit `n`$\geq 7$, on définit le tableau `tab` de taille `n` par :

— `tab.(0) = n-4`

— `tab.(1) = 2`

— `tab.(2) = 1`

— `tab.(n-4) = 1`

— `tab.(i) = 0` si `i` $\notin \{0, 1, 2, n-4\}$

Prouver que `tab` est autoréférent

On vérifie :

— `tab.(0) = n-4` et toutes les cases du tableau valent `0` sauf 4 cases, celles d'indices 0 (car $n-4 \neq 0$), 1, 2 et $n-4$.

— `tab.(1) = 2` et `1` apparaît bien deux fois dans le tableau aux indices 2 et $n-4$. (car $n-4 \neq 1$)

— `tab.(2) = 1` et `2` apparaît bien une fois dans le tableau (à l'indice 1)

— `tab.(n-4) = 1` et `n-4` apparaît bien une seule fois dans le tableau car comme $n \leqslant 7$, $n-4 \leqslant 3$.

— `tab.(i) = 0` si `i` $\notin \{0, 1, 2, n-4\}$ et aucune de ces valeurs n'apparaît dans le tableau

Donc ce tableau est bien autoréférent.

5. Ecrire une fonction `est_autoreferent` qui prend en argument un tableau d'entiers (et sa taille) et renvoie `true` si ce tableau est autoréférent et `false` sinon. On attend une complexité en $\mathcal{O}(n)$ où $n$ est la taille du tableau c'est à dire qu'on veut parcourir une seule fois le tableau.

```
bool est_autoreferent(int tab[], int n)
{
    // On initialise les occurrences de toutes les valeurs à 0
    int occ[n];
    for (int k = 0; k < n; k++)
    {
        occ[k] = 0;
    }
    // On parcourt en incrémentant les occurrences
    for (int i = 0; i < n; i++)
    {
        occ[tab[i]]++;
    }
    // On teste si le tableau est autoréférent
    for (int i = 0; i < n; i++)
    {
        if (occ[i] != tab[i])
        {
            return false;
        }
    }
    return true;
}
```

On cherche maintenant à construire un tableau autoréférent de taille `n` en utilisant un algorithme de recherche par retour sur trace (*backtracking*) qui valide une solution partielle construite jusqu'à un index `i` donné, on propose pour cela la fonction récursive suivante qui utilise la fonction de validation partielle `valide` qui sera écrite plus loin et qui essaye d'affecter une valeur valide à l'indice `i` puis de poursuivre la construction du tableau :

```
1  bool autoreferent(int tab[], int n, int i)
2  {
3      if (i == n)
4      {
5          return est_autoreferent(tab,n);
6      }
7      for (int k = .......)
8      {
9          tab[i] = k;
10         if (valide(tab, n, i))
11         {
12             if (autoreferent(........))
13             {
14                 return .....;
15             }
16         }
17     }
18     return ......;
19 }
```

6. Compléter les lignes 7, 12, 14, et 18 de la fonction précédente.

```
1  bool autoreferent(int tab[], int n, int i)
2  {
3      if (i == n)
4      {
5          return est_autoreferent(tab,n);
6      }
7      for (int k = 0; k <= n; k++)
8      {
9          tab[i] = k;
10         if (valide(tab, n, i))
11         {
12             if (autoreferent(tab, n, i + 1))
13             {
14                 return true;
15             }
16         }
17     }
18     return false;
19 }
```

7. On sait que la somme des éléments d'un tableau autoréférent de taille `n` est `n`. D'autre part, si après avoir affecté la case d'indice `i`, il y déjà strictement plus d'occurrences d'une valeur `k` comprise entre 0 et `i` que la valeur de `t[k]` alors la solution partielle n'est pas valide. En déduire une fonction `valide` de signature `bool valide(int tab[], int n, int i)` qui prend en argument un tableau d'entiers, sa taille et un indice `i` et qui renvoie `true` si la ce tableau peut-être complété en un tableau autoréférent.

```cpp
bool valide(int tab[], int n, int i)
{
    int s = 0;
    for (int k = 0; k < i; k++)
    {
        s += tab[k];
    }
    if (s > n)
    {
        return false;
    }
    int occ[i];
    for (int k = 0; k < i; k++)
    {
        occ[k] = 0;
    }
    for (int k = 0; k < i; k++)
    {
        occ[tab[k]]++;
    }
    for (int k = 0; k < i; k++)
    {
        if (occ[k] > tab[k])
        {
            return false;
        }
    }
    return true;
}
```

8. Proposer et tester une nouvelle vérification permettant d'écarter plus rapidement les solutions non valides. On pourra s'intéresser au nombre de valeurs à compléter contenant une valeur non nulle.