Math93.com

BAC NSI - Correction Amérique Nord - 2026 - Sujet 1 - NSI

Thèmes des exercices (sur 20 points)

- Exercice 1 (6 points) : Programmation orientée objet, récursivité et algorithme min-max.
- Exercice 2 (6 points) : Réseaux, routage, files et programmation orientée objet.
- Exercice 3 (8 points) : Bases de données, récursivité et programmation dynamique.
- Lien vers le sujet

Exercice 1. Programmation orientée objet, récursivité et algorithme min-max. 6 points

Cet exercice porte sur : Programmation orientée objet, récursivité et algorithme min-max.

Le jeu puissance 4 se joue à deux joueurs dans une grille de 6 lignes et 7 colonnes. Une couleur de pion, blanc ou noir, est attribuée à chaque joueur. Les joueurs jouent à tour de rôle. Lorsqu'il joue, un joueur choisit une colonne dans laquelle il introduit un pion de sa couleur. La grille de jeu est placée verticalement de sorte que le pion introduit tombe dans la colonne choisie et bute soit sur le bas de la grille, soit sur un autre pion déjà placé. Le but du jeu pour chaque joueur est d'aligner au moins quatre pions de sa couleur dans n'importe quelle direction : horizontalement, verticalement ou en diagonale. Le premier à y parvenir a gagné.

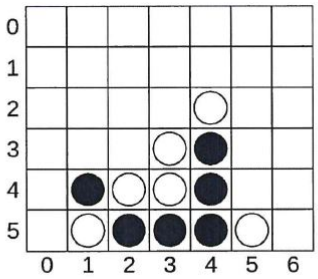


Figure 1. Une partie gagnée par le joueur blanc

Figure 1. Une partie gagnée par le joueur blanc

Le but de l'exercice est d'implémenter un algorithme appelé min-max permettant à un joueur d'optimiser ses chances de victoire selon un principe simple : à chaque coup possible du joueur, on associe un score calculé en fonction de tous les futurs coups possibles, les siens, mais aussi ceux de son adversaire. Le coup qui a le meilleur score est celui qu'il faut choisir.
Pour simuler l'ensemble des coups possibles, on utilise un arbre dont les nœuds correspondent à un coup. Chaque fils d'un nœud correspond à une possibilité de coup suivant le coup considéré.

Partie A : grille de jeu et score associé

Dans la suite, les deux joueurs seront notés 1 et 2.
Pour gérer la grille de jeu, on va écrire une classe `Grille`. L'unique attribut d'un objet instance de la classe `Grille` est un tableau nommé `grille` de 6 lignes et 7 colonnes, représenté en Python par une liste de listes. Ce tableau contient des entiers qui ne peuvent prendre que trois valeurs : 0, 1 ou 2.
La valeur 0 signifie que la case du jeu correspondante est vide. La valeur 1 indique qu'un pion du joueur 1 se trouve dans la case et la valeur 2 qu'un pion du joueur 2 s'y trouve. La convention de numérotation des lignes et des colonnes dans le tableau est présentée sur la figure 1.

1. Écrire la méthode `__init__(self)` de la classe `Grille` qui définit l'attribut `grille` comme un tableau rempli de 0.

Corrigé

Constructeur et attribut d'instance

Dans une classe Python, la méthode `__init__` est appelée automatiquement lors de la création d'un nouvel objet.

Le paramètre `self` désigne l'objet en cours de construction. Pour créer un attribut d'instance, on écrit :

`self.nom_attribut = valeur.`

On doit créer un tableau de 6 lignes et 7 colonnes. Chaque case doit contenir initialement la valeur 0.

Une bonne manière de construire une liste de listes indépendantes consiste à utiliser une liste en compréhension.

La réponse Python est donnée ci-dessous.

```
class Grille:
    def __init__(self):
        self.grille = [
            [0 for colonne in range(7)]
            for ligne in range(6)
        ]
```

Remarque

On évite l'écriture `[[0]*7]*6`, car elle crée plusieurs références vers une même liste interne. Modifier une ligne modifierait alors les autres lignes, ce qui ne convient pas pour représenter une grille.

2. Recopier et compléter les lignes 4, 6, 7 et 9 de la méthode `joue(self, colonne, joueur)` suivante de la classe `Grille`. Son but est de tenter de placer un pion du joueur `joueur` dans la colonne dont le numéro est `colonne`. Si le coup est possible, c'est-à-dire si la colonne ne contient pas déjà six pions, alors le pion est placé dans la grille au bon endroit et la fonction renvoie `True`. Si le coup n'est pas possible, la fonction renvoie simplement `False`.

Remarque importante : la recherche d'une case où placer le pion se fait de bas en haut.

```
def joue(self, colonne, joueur):
    ligne = 5  # on part de la rangee la plus basse
    while ligne != -1 and self.grille[ligne][colonne] != 0:
        ligne = ...
    if ligne != -1:
        self.grille[ligne][colonne] = ...
        return ...
    else:
        return ...
```

Corrigé

Parcours d'une colonne dans une grille

Dans la grille, les lignes sont numérotées de 0 à 5.

Comme le pion tombe vers le bas, on cherche la première case vide en partant de la ligne 5, puis en remontant jusqu'à la ligne 0.

La variable `ligne` est initialisée à 5, car on commence par la rangée la plus basse.

Tant que la ligne existe encore et que la case observée n'est pas vide, on remonte d'une ligne avec :

`ligne = ligne - 1.`

Si une case vide est trouvée, on y place le numéro du joueur, puis on renvoie `True`. Sinon, la colonne est pleine et on renvoie `False`.

La méthode complétée est donnée ci-dessous.

```
def joue(self, colonne, joueur):
    ligne = 5  # on part de la rangee la plus basse
    while ligne != -1 and self.grille[ligne][colonne] != 0:
        ligne = ligne - 1
    if ligne != -1:
        self.grille[ligne][colonne] = joueur
        return True
    else:
        return False
```

Voici un exemple de tableau représentant la grille de jeu obtenue à la fin du troisième coup :

```
[[0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 1, 0, 0, 0],
 [0, 0, 1, 2, 0, 0, 0]]
```

3. Écrire le code Python nécessaire pour créer cette grille de jeu en utilisant uniquement les méthodes de la classe `Grille`. Elle sera stockée dans une variable `jeu1`.

Corrigé

On doit créer un objet de la classe `Grille`, puis jouer les trois coups permettant d'obtenir le tableau proposé.

Dans la grille finale :

- le joueur 1 a placé un pion dans la colonne 2 ;
- le joueur 2 a placé un pion dans la colonne 3 ;
- le joueur 1 a placé un second pion dans la colonne 3, au-dessus du pion du joueur 2.

Il suffit donc de jouer successivement dans les colonnes 2, 3, puis 3.

Le code est donné ci-dessous.

```
jeu1 = Grille()
jeu1.joue(2, 1)
jeu1.joue(3, 2)
jeu1.joue(3, 1)
```

À présent, on va écrire une méthode de la classe `Grille` qui associe à une grille un score en fonction des pions déjà placés.

On suppose qu'on a déjà écrit une fonction `valeur_case(ligne, colonne)` qui renvoie le nombre d'alignements de quatre cases contenant la case `(ligne, colonne)`. Par exemple, l'appel `valeur_case(0, 1)` renvoie 4. En effet, il y a quatre alignements de quatre cases qui contiennent la case (0, 1) et qui sont représentés à la figure 2.

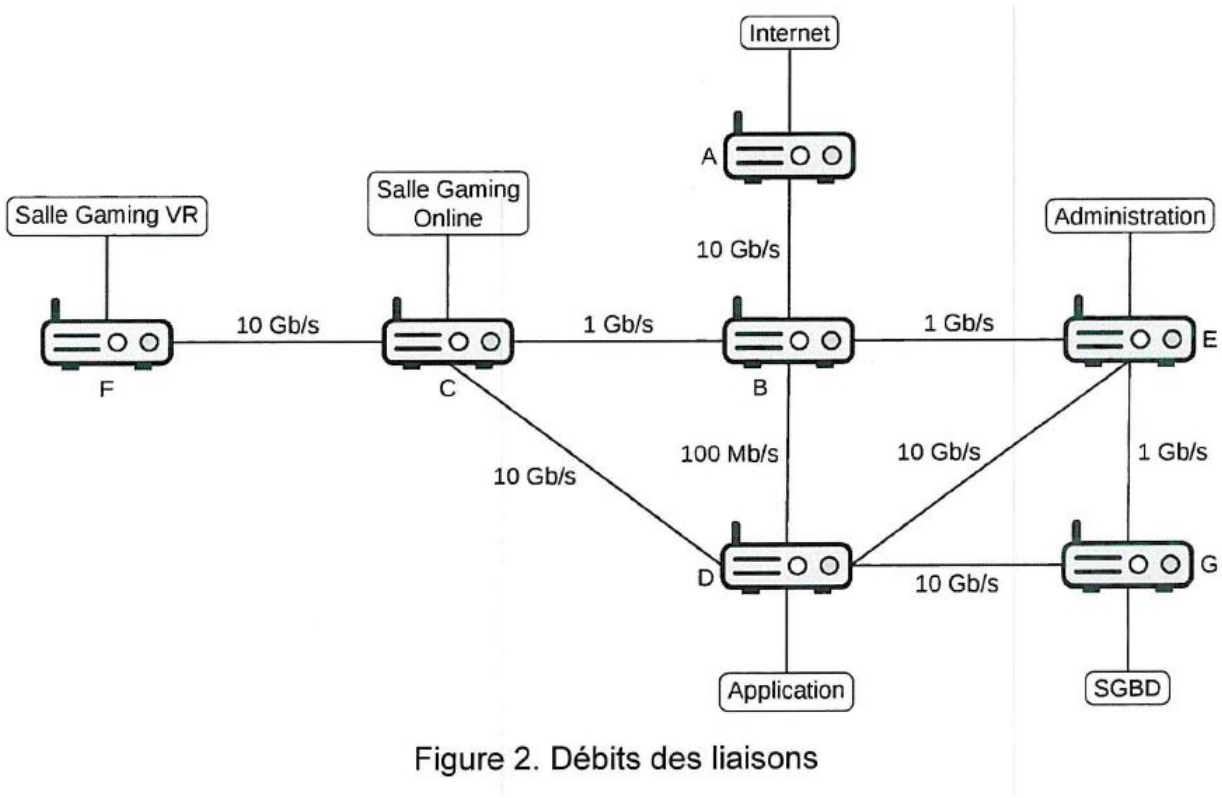


Figure 2. Débits des liaisons

Figure 2. Alignements contenant la case (0, 1)

Le score d'une grille est calculé en additionnant les valeurs de toutes les cases occupées par un pion du joueur 2 et en soustrayant les valeurs de toutes les cases occupées par un pion du joueur 1, la valeur d'une case étant obtenue à l'aide de la fonction `valeur_case`.

4. Donner le score associé à la grille de jeu `jeu1` donnée à la question 3 en expliquant bien le calcul effectué.

Corrigé

Score d'une grille

Le score d'une grille est obtenu :

- en ajoutant les valeurs des cases occupées par le joueur 2 ;
- en soustrayant les valeurs des cases occupées par le joueur 1.

Dans la grille `jeu1`, les cases occupées sont :

$$(5,2), \quad (5,3), \quad (4,3).$$

Le joueur 1 occupe les cases :

$$(5,2) \quad \text{et} \quad (4,3),$$

tandis que le joueur 2 occupe la case :

$$(5,3).$$

On obtient :

$$\texttt{valeur\_case(5,3)} = 7,$$

$$\texttt{valeur\_case(5,2)} = 5,$$

et :

$$\texttt{valeur\_case(4,3)} = 10.$$

Ainsi :

$$\begin{aligned} \text{score}(\texttt{jeu1}) &= 7 - 5 - 10 \\ &= -8. \end{aligned}$$

Donc :

$$\boxed{\text{score}(\texttt{jeu1}) = -8.}$$

5. Écrire la méthode `score(self)` qui renvoie le score associé à la grille de jeu représentée par l'objet.

Corrigé

Pour calculer le score, on parcourt toutes les cases de la grille.

- Si une case contient 2, elle est occupée par le joueur 2, donc on ajoute la valeur de cette case.
- Si une case contient 1, elle est occupée par le joueur 1, donc on soustrait la valeur de cette case.
- Si une case contient 0, elle est vide et ne contribue pas au score.

La méthode complète est donnée ci-dessous.

```
def score(self):
    total = 0
    for ligne in range(6):
        for colonne in range(7):
            if self.grille[ligne][colonne] == 2:
                total = total + valeur_case(ligne, colonne)
            elif self.grille[ligne][colonne] == 1:
                total = total - valeur_case(ligne, colonne)
    return total
```

Partie B : algorithme min-max et création de l'arbre de coups

Dans la suite, on associe un arbre de coups à un joueur et à une grille de jeu. Il permet de simuler tous les coups possibles pour le joueur et son adversaire et de calculer les scores associés à chaque coup en respectant l'algorithme min-max décrit ci-dessous.

- On ne calcule qu'un nombre maximum donné de coups à l'avance. On note `niveau_max` ce nombre qui correspond donc au niveau de profondeur maximale atteint par l'arbre, avec la convention que sa racine est au niveau 0. `niveau_max` est une valeur qu'on considérera comme une constante accessible en tant que variable globale.
- Si un nœud correspond à un coup gagnant pour un des deux joueurs, son score est égal à

$$-\big(100+10\times(\texttt{niveau\_max}-\texttt{niveau})\big)$$

pour le joueur 1 et

$$100+10\times(\texttt{niveau\_max}-\texttt{niveau})$$

pour le joueur 2, où `niveau` désigne le niveau de profondeur du nœud dans l'arbre de coups.
- Si un nœud se trouve au niveau `niveau_max`, son score est égal au score de la grille de jeu, calculé grâce à la méthode `score` écrite dans la partie A.
- Enfin, dans tous les autres cas, le score d'un nœud est le minimum des scores de ses fils s'il s'agit d'un coup du joueur 1 et le maximum des scores de ses fils s'il s'agit d'un coup du joueur 2.

Afin d'optimiser ses chances de victoire, quand c'est à son tour de jouer, le joueur 1 doit choisir le nœud de niveau 1 correspondant au coup ayant le score le plus petit tandis qu'à son tour, le joueur 2 doit choisir le coup ayant le score le plus grand.
Pour construire l'arbre de coups, on va utiliser une classe `Noeud`.
Une instance de la classe `Noeud` a trois attributs :

- `colonne`, qui vaut soit -1, soit le numéro de la colonne où le coup est joué, de 0 à 6 ;
- `score`, qui correspond au score associé au coup ;
- `suivants`, qui est la liste de ses nœuds fils.

La racine d'un arbre de coups est un nœud ne représentant pas un coup. Son attribut `colonne` est initialisé à -1, ses fils représentent tous les premiers coups possibles pour le joueur 1.

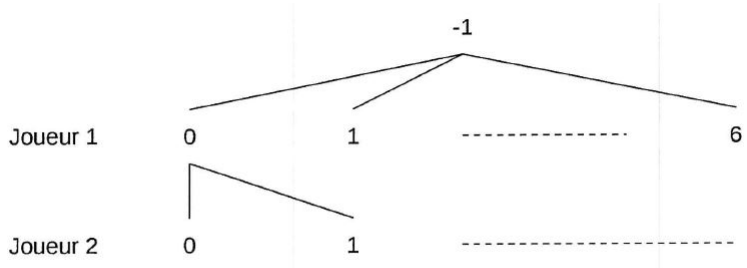


Figure 3. Schéma partiel de l'arbre de coups avec niveau_max = 2 où l'on a indiqué la valeur de l'attribut colonne

Figure 3. Schéma partiel de l'arbre de coups avec `niveau_max = 2` où l'on a indiqué la valeur de l'attribut `colonne`

6. Compléter la méthode `__init__(self, colonne)` de la classe `Noeud` permettant de créer un nœud de l'arbre symbolisant un coup joué dans la colonne `colonne` avec un score nul et aucun nœud fils.

```
class Noeud:
    def __init__(self, colonne):
        ...
        ...
        ...
```

Corrigé

Un nœud doit mémoriser trois informations :

- la colonne dans laquelle le coup est joué ;
- le score associé au coup, initialement nul ;
- la liste des nœuds fils, initialement vide.

La méthode complétée est donnée ci-dessous.

```
class Noeud:
    def __init__(self, colonne):
        self.colonne = colonne
        self.score = 0
        self.suivants = []
```

7. Écrire une méthode `colonne_score_min` de la classe `Noeud` qui renvoie le couple `(colonne, score)` correspondant au numéro de la colonne et au score d'un des fils du nœud pour lequel le score est le plus petit. On suppose que le nœud a au moins un fils.

Corrigé

Recherche d'un minimum

Pour rechercher un minimum dans une liste non vide, on peut :

- initialiser le minimum avec le premier élément de la liste ;
- parcourir les autres éléments ;
- mettre à jour le minimum dès qu'un élément plus petit est rencontré.

On cherche ici, parmi les fils du nœud, un fils dont le score est minimal. On renvoie ensuite le couple formé par la colonne de ce fils et son score.

La méthode est donnée ci-dessous.

```
def colonne_score_min(self):
    noeud_min = self.suivants[0]
    for noeud in self.suivants:
        if noeud.score < noeud_min.score:
            noeud_min = noeud
    return (noeud_min.colonne, noeud_min.score)
```

On suppose dans la suite qu'on a écrit de même une méthode `colonne_score_max` qui renvoie un couple `(colonne, score)` correspondant au numéro de la colonne et au score d'un des fils du nœud pour lequel le score est le plus grand.

On suppose qu'on a ajouté les méthodes suivantes à la classe `Grille` :

- `gagnant(self)`, renvoyant le numéro du joueur gagnant, 1 ou 2, s'il existe un alignement de quatre de ses pions dans la grille représentée par l'objet, ou 0 s'il n'y a aucun gagnant. On suppose qu'il ne peut y avoir qu'un gagnant ;
- `copie_grille(self)`, renvoyant une grille de jeu, copie exacte de la grille représentée par l'objet. Ainsi, si on modifie une copie de la grille, la grille n'est pas modifiée.

8. Recopier et compléter le code de la méthode `calcule_score(self, niveau, joueur, grille)` de la classe `Noeud` suivante, qui attribue un score au nœud représenté par l'objet conformément à l'algorithme min-max décrit plus haut.

   `niveau` est le niveau de profondeur du nœud dans l'arbre de coups. `joueur` est le numéro du joueur dont le nœud représente le coup. `grille` est un objet `Grille` représentant le jeu juste après que le coup correspondant au nœud a été joué.

```
def calcule_score(self, niveau, joueur, grille):
    g = grille.gagnant()
    if g == 1:
        self.score = ...
    elif g == 2:
        self.score = ...
    elif niveau == niveau_max:
        self.score = ...
    else:
        for colonne in range(7):
            grille2 = grille.copie_grille()
            if grille2.joue(colonne, joueur):
                nouveau_noeud = ...
                self.suivants.append(...)
                nouveau_noeud.calcule_score(...)
        if joueur == 1:
            self.score = ...
        else:
            self.score = ...
```

Corrigé

Principe de l'algorithme min-max

L'algorithme min-max attribue un score aux nœuds d'un arbre de coups.

- Si la position est gagnante, on attribue un score très favorable au joueur gagnant.
- Si la profondeur maximale est atteinte, on évalue directement la grille.
- Sinon, on propage les scores depuis les fils : le joueur 1 minimise le score, tandis que le joueur 2 le maximise.

On suit exactement les règles données dans l'énoncé.

- Si le joueur 1 gagne, le score est :

$$-\big(100+10\times(\texttt{niveau\_max}-\texttt{niveau})\big).$$

- Si le joueur 2 gagne, le score est :

$$100+10\times(\texttt{niveau\_max}-\texttt{niveau}).$$

- Si la profondeur maximale est atteinte, le score est celui de la grille.

- Sinon, on crée les fils correspondant aux coups possibles, puis on applique la règle du minimum ou du maximum selon le joueur.

La méthode complétée est donnée ci-dessous.

```
def calcule_score(self, niveau, joueur, grille):
    g = grille.gagnant()
    if g == 1:
        self.score = -(100 + 10 * (niveau_max - niveau))
    elif g == 2:
        self.score = 100 + 10 * (niveau_max - niveau)
    elif niveau == niveau_max:
        self.score = grille.score()
    else:
        for colonne in range(7):
            grille2 = grille.copie_grille()
            if grille2.joue(colonne, joueur):
                nouveau_noeud = Noeud(colonne)
                self.suivants.append(nouveau_noeud)
                nouveau_noeud.calcule_score(
                    niveau + 1,
                    3 - joueur,
                    grille2
                )
        if joueur == 1:
            self.score = self.colonne_score_min()[1]
        else:
            self.score = self.colonne_score_max()[1]
```

9. Indiquer pourquoi il n'est pas réaliste d'utiliser cet algorithme pour explorer l'ensemble des parties, ce qui correspond à la valeur 42 pour `niveau_max`.

Corrigé

Une grille de puissance 4 comporte :

$$6 \times 7 = 42$$

cases. Explorer toutes les parties revient donc à envisager jusqu'à 42 coups.

À chaque coup, il peut y avoir jusqu'à 7 possibilités, correspondant aux 7 colonnes de la grille. Le nombre de branches à explorer peut donc devenir extrêmement grand.

Un ordre de grandeur grossier du nombre de suites de coups possibles est :

$$7^{42}.$$

Or ce nombre est gigantesque :

$$7^{42} \approx 3,1 \times 10^{35}.$$

Il n'est donc pas réaliste d'explorer l'ensemble de l'arbre des parties, car cela demanderait un temps de calcul et une quantité de mémoire beaucoup trop importants.

L'exploration complète de toutes les parties n'est pas réaliste.

Partie C : choix du meilleur coup à jouer

10. Utiliser les fonctions précédentes et la classe `Noeud` afin d'écrire une fonction `choisit_coup(grille, joueur)` prenant en arguments une grille de jeu et le numéro d'un joueur et renvoyant le numéro de la colonne où devrait jouer le joueur pour optimiser ses chances de victoire selon le principe de l'algorithme min-max.

Corrigé

On commence par créer la racine de l'arbre de coups. Cette racine ne correspond à aucun coup joué : son attribut `colonne` vaut donc -1.

On calcule ensuite récursivement les scores de l'arbre à partir de cette racine.

Enfin :

- si le joueur est le joueur 1, il choisit la colonne associée au score minimal ;
- si le joueur est le joueur 2, il choisit la colonne associée au score maximal.

La fonction renvoie donc le numéro de la colonne choisie.

```
def choisit_coup(grille, joueur):
    racine = Noeud(-1)
    racine.calcule_score(0, joueur, grille)
    if joueur == 1:
        colonne, score = racine.colonne_score_min()
    else:
        colonne, score = racine.colonne_score_max()
    return colonne
```

Exercice 2. Réseaux, routage, files et programmation orientée objet — 6 points

Cet exercice porte sur l'architecture matérielle (réseau), les structures de données et la programmation orientée objet.

L'entreprise Gamerzz propose à ses clients plusieurs services :

- une salle permettant de jouer à des jeux vidéo en ligne, appelée « Salle Gaming Online » ;
- une salle permettant de jouer en réalité virtuelle, appelée « Salle Gaming VR » ;
- un site permettant de réserver, d'organiser des événements et de gérer les classements des joueurs en réalité virtuelle.

On donne ci-dessous un schéma du réseau de l'entreprise Gamerzz. Les différents sous-réseaux sont indiqués avec leur notation CIDR.

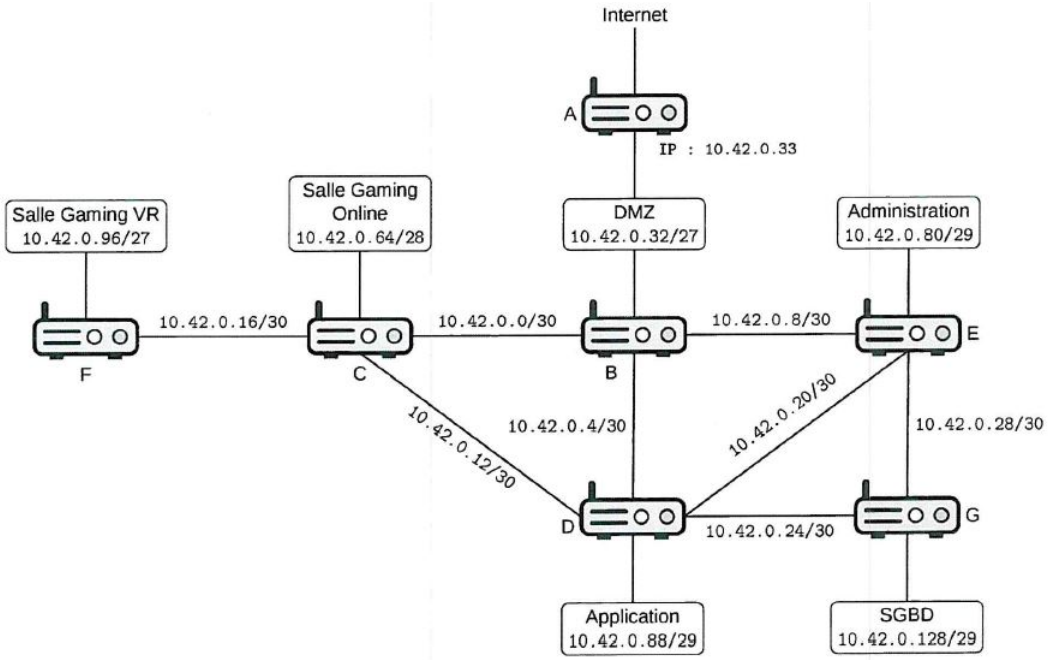


Figure 1. Réseau de l'entreprise Gamerzz

La notation `a.b.c.d/n`, appelée notation CIDR, signifie que les n premiers bits à gauche de l'adresse IP représentent la partie « réseau ». Les bits situés à droite représentent la partie « machine ».
L'adresse IPv4 dont tous les bits de la partie « machine » sont à 0 est appelée « adresse du réseau ». L'adresse IPv4 dont tous les bits de la partie « machine » sont à 1 est appelée « adresse de diffusion ». Les autres adresses peuvent être attribuées à des machines, telles que des routeurs, des serveurs ou des ordinateurs.

1. Donner le nombre d'adresses IP qui peuvent être attribuées à des machines dans le réseau « Administration », ainsi qu'une adresse possible pour le routeur E.

Corrigé

Notation CIDR et adresses utilisables

En notation CIDR, un réseau noté `a.b.c.d/n` possède :

$$32 - n$$

bits pour la partie « machine ».
Il contient donc :

$$2^{32-n}$$

adresses au total.
Parmi ces adresses, deux ne sont pas attribuables à des machines :

- l'adresse du réseau, dont tous les bits de la partie machine valent 0 ;
- l'adresse de diffusion, dont tous les bits de la partie machine valent 1.

D'après la figure, le réseau « Administration » est :

`10.42.0.80/29`.

Il reste donc :

$$32 - 29 = 3$$

bits pour la partie machine.
Le nombre total d'adresses dans ce sous-réseau est :

$$2^3 = 8.$$

On retire l'adresse du réseau et l'adresse de diffusion :

$$8 - 2 = 6.$$

Il y a donc :

$$\boxed{6}$$

adresses IP attribuables à des machines dans le réseau « Administration ».
Le réseau `10.42.0.80/29` contient les adresses allant de :

`10.42.0.80` à `10.42.0.87`.

L'adresse `10.42.0.80` est l'adresse du réseau et l'adresse `10.42.0.87` est l'adresse de diffusion.
Les adresses utilisables sont donc :

`10.42.0.81`, `10.42.0.82`, ..., `10.42.0.86`.

Une adresse possible pour le routeur E est donc par exemple :

$$\boxed{\texttt{10.42.0.81.}}$$

Des tentatives de téléchargements non autorisés sont détectées depuis l'adresse IP `10.42.0.70`.

2. Indiquer dans quel réseau se situe la machine correspondante.

Corrigé

L'adresse IP concernée est :

`10.42.0.70`.

D'après la figure, le réseau « Salle Gaming Online » est :

`10.42.0.64/28`.

Dans un réseau en `/28`, il reste :

$$32 - 28 = 4$$

bits pour la partie machine.
Le réseau contient donc :

$$2^4 = 16$$

adresses au total.
Le réseau `10.42.0.64/28` contient les adresses de :

`10.42.0.64` à `10.42.0.79`.

Comme :

$$64 \leq 70 \leq 79,$$

l'adresse `10.42.0.70` appartient à ce réseau.
Ainsi :

la machine se situe dans le réseau « Salle Gaming Online ».

Les réseaux qui interconnectent les routeurs sont en `/30`. Cela permet d'attribuer une adresse exactement à deux machines, les deux autres adresses étant l'adresse du réseau et l'adresse de diffusion.
On choisit, sur un tel réseau, d'attribuer les adresses IP des routeurs dans le même ordre que celui des lettres qui les désignent sur la figure 1.

3. Donner les adresses IP attribuées, selon ce principe, aux routeurs C et F dans le réseau `10.42.0.16/30`, ainsi que les adresses IP attribuées aux routeurs C et D dans le réseau `10.42.0.12/30`.

Corrigé

Réseau en `/30`

Un réseau en `/30` contient :

$$2^{32-30} = 2^2 = 4$$

adresses.
Parmi ces quatre adresses :

- la première est l'adresse du réseau ;
- la dernière est l'adresse de diffusion ;
- les deux adresses intermédiaires sont attribuables aux machines.

- Pour le réseau :

`10.42.0.16/30`,

les quatre adresses sont :

$$\texttt{10.42.0.16},\ \texttt{10.42.0.17},\ \texttt{10.42.0.18},\ \texttt{10.42.0.19}.$$

L'adresse `10.42.0.16` est l'adresse du réseau et l'adresse `10.42.0.19` est l'adresse de diffusion.
Les deux adresses attribuables sont donc :

$$\texttt{10.42.0.17} \quad \text{et} \quad \texttt{10.42.0.18}.$$

Les routeurs concernés sont C et F. Comme on attribue les adresses dans l'ordre alphabétique des lettres des routeurs, on obtient :

$$\boxed{\text{C} : \texttt{10.42.0.17} \quad \text{et} \quad \text{F} : \texttt{10.42.0.18}.}$$

- Pour le réseau :
$$\texttt{10.42.0.12/30},$$
les quatre adresses sont :
$$\texttt{10.42.0.12},\ \texttt{10.42.0.13},\ \texttt{10.42.0.14},\ \texttt{10.42.0.15}.$$
L'adresse `10.42.0.12` est l'adresse du réseau et l'adresse `10.42.0.15` est l'adresse de diffusion.
Les deux adresses attribuables sont donc :
$$\texttt{10.42.0.13} \quad \text{et} \quad \texttt{10.42.0.14}.$$
Les routeurs concernés sont C et D. On attribue donc :
$$\boxed{\text{C} : \texttt{10.42.0.13} \quad \text{et} \quad \text{D} : \texttt{10.42.0.14}.}$$

On choisit de configurer les routeurs suivant le protocole RIP. Le protocole RIP permet de minimiser le nombre de routeurs traversés par les paquets.
Voici la table de routage du routeur B.

Routeur B		
Réseau	Passerelle	Nombre de sauts
DMZ	connecté	0
Gaming Online	`10.42.0.2`	1
Internet	`10.42.0.33`	1
Gaming VR	`10.42.0.2`	2
Administration	`10.42.0.10`	1
Application	`10.42.0.6`	1
SGBD	`10.42.0.6`	2

4. Donner une table possible pour le routeur C, en suivant le protocole RIP.

Corrigé

Protocole RIP

Le protocole RIP choisit une route en minimisant le nombre de sauts.
Un saut correspond au passage par un routeur. La passerelle indiquée dans une table de routage est l'adresse IP du prochain routeur auquel transmettre le paquet.

On construit une table de routage possible pour le routeur C.
Le routeur C est directement relié :

- au réseau « Gaming Online » ;
- au routeur B ;
- au routeur D ;
- au routeur F.

Sur le réseau `10.42.0.0/30`, les routeurs B et C ont respectivement les adresses :

B : `10.42.0.1` et C : `10.42.0.2`.

D'après la question précédente, on a aussi :

D : `10.42.0.14` et F : `10.42.0.18`.

Une table possible pour le routeur C est donc :

Routeur C		
Réseau	Passerelle	Nombre de sauts
Gaming Online	connecté	0
Gaming VR	`10.42.0.18`	1
DMZ	`10.42.0.1`	1
Internet	`10.42.0.1`	2
Administration	`10.42.0.1`	2
Application	`10.42.0.14`	1
SGBD	`10.42.0.14`	2

Remarque

Pour atteindre le réseau « Administration », il existe aussi un chemin de même longueur en passant par le routeur D puis le routeur E. Une autre table correcte pourrait donc indiquer la passerelle `10.42.0.14` pour ce réseau, avec toujours 2 sauts.

On indique maintenant les débits des liaisons entre routeurs.

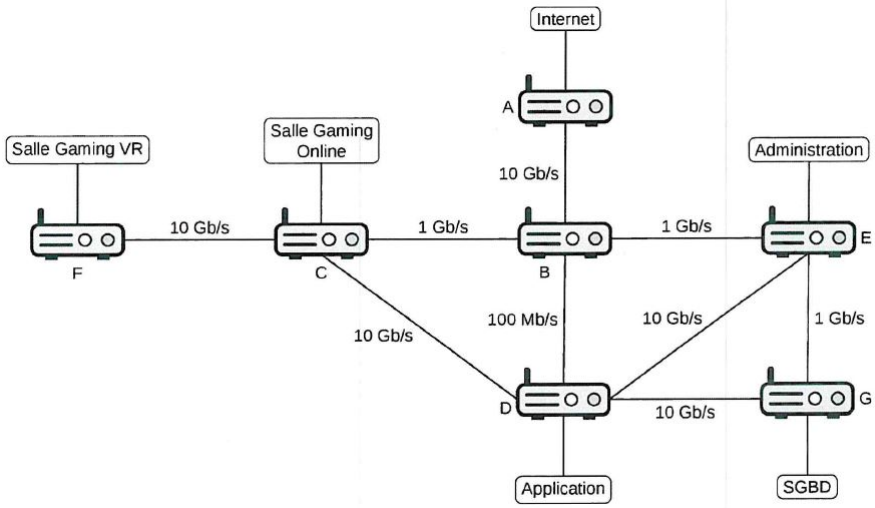


Figure 2. Débits des liaisons

On précise que :

- le coût d'une liaison est donné par
$$\frac{10^{10}}{d},$$
où d est le débit de la liaison en bits par seconde ;
- le protocole OSPF fait transiter les informations de routeur en routeur en minimisant le coût total de leur acheminement.

5. Donner, pour chaque débit présent en figure 2, le coût de la liaison.

Corrigé

On utilise la formule :
$$\text{coût} = \frac{10^{10}}{d},$$
où d est le débit de la liaison en bits par seconde.
Les débits présents sur la figure sont :
$$10\text{ Gb/s}, \qquad 1\text{ Gb/s}, \qquad 100\text{ Mb/s}.$$

- Pour une liaison à 10 Gb/s :
$$10\text{ Gb/s} = 10 \times 10^9 = 10^{10}\text{ bits/s}.$$
Donc :
$$\frac{10^{10}}{10^{10}} = 1.$$
- Pour une liaison à 1 Gb/s :
$$1\text{ Gb/s} = 10^9\text{ bits/s}.$$
Donc :
$$\frac{10^{10}}{10^9} = 10.$$
- Pour une liaison à 100 Mb/s :
$$100\text{ Mb/s} = 100 \times 10^6 = 10^8\text{ bits/s}.$$

Donc :

$$\frac{10^{10}}{10^{8}} = 100.$$

Ainsi :

Débit	Coût
10 Gb/s	1
1 Gb/s	10
100 Mb/s	100

Une information venue d'un client extérieur, par exemple depuis Internet, est acheminée vers le réseau « Application » pour être traitée.

6. Donner un des ordres possibles des routeurs par lesquels cette information transite à partir du routeur A en suivant le protocole OSPF.

Corrigé

Principe du protocole OSPF

Le protocole OSPF choisit une route qui minimise le coût total du chemin.

Le coût total d'un chemin est la somme des coûts des liaisons empruntées.

Le réseau « Application » est relié au routeur D. On cherche donc un chemin de coût minimal entre le routeur A et le routeur D.

D'après la figure 2 :

- la liaison A–B est à 10 Gb/s, donc son coût est 1 ;
- la liaison B–C est à 1 Gb/s, donc son coût est 10 ;
- la liaison C–D est à 10 Gb/s, donc son coût est 1.

Le chemin :

$$A \longrightarrow B \longrightarrow C \longrightarrow D$$

a donc pour coût :

$$1 + 10 + 1 = 12.$$

Le chemin direct passant par la liaison B–D utilise une liaison à 100 Mb/s, donc de coût 100. Le chemin :

$$A \longrightarrow B \longrightarrow D$$

a donc pour coût :

$$1 + 100 = 101.$$

Il n'est donc pas optimal.

Un ordre possible des routeurs empruntés selon OSPF est :

$$\boxed{A \longrightarrow B \longrightarrow C \longrightarrow D.}$$

Remarque

Le chemin

$$A \longrightarrow B \longrightarrow E \longrightarrow D$$

a également un coût total égal à 12. Il constitue donc aussi une réponse possible.

On s'intéresse désormais à la gestion des paquets TCP par un routeur.

7. Rappeler si les données d'un segment TCP sont contenues dans un paquet IP, ou bien si les données d'un paquet IP sont contenues dans un segment TCP.

Corrigé

Encapsulation TCP/IP

Dans le modèle TCP/IP, les données sont encapsulées couche après couche.

Un segment TCP est placé dans un paquet IP. On dit que le paquet IP encapsule le segment TCP.

Les données d'un segment TCP sont contenues dans un paquet IP.

Ainsi :

un paquet IP contient un segment TCP.

Lorsqu'un routeur reçoit un paquet TCP, il le place dans une file d'attente avant de pouvoir le transmettre.

8. Expliquer pourquoi il est plus intéressant dans ce cas précis d'utiliser une file plutôt qu'une pile.

Corrigé

File et pile

Une file est une structure de données de type FIFO, pour « First In, First Out » : le premier élément entré est le premier à sortir.

Une pile est une structure de données de type LIFO, pour « Last In, First Out » : le dernier élément entré est le premier à sortir.

Dans le cas d'un routeur, on souhaite transmettre les paquets dans leur ordre d'arrivée, afin de conserver une gestion cohérente et équitable du trafic.

Une file est donc adaptée, car le premier paquet reçu est aussi le premier paquet transmis.

Au contraire, avec une pile, les paquets les plus récents seraient transmis avant les plus anciens. Des paquets plus anciens pourraient donc attendre très longtemps si de nouveaux paquets arrivent en permanence.

Ainsi :

il est plus intéressant d'utiliser une file, car elle respecte l'ordre d'arrivée des paquets.

On modélise les files en Python à l'aide des fonctions suivantes :

- `cree_file` : renvoie une file vide ;
- `est_vide` : renvoie le booléen indiquant si la file `f` passée en paramètre est vide ;
- `enfile` : prend en paramètres une file `f` et un élément `x`, puis ajoute `x` dans `f` ;
- `defile` : prend en paramètre une file `f` non vide, supprime l'élément de `f` le plus anciennement ajouté et renvoie sa valeur.

On donne un exemple d'utilisation de ces fonctions.

```
>>> f = cree_file()
>>> est_vide(f)
True
>>> enfile(f, 0)
>>> enfile(f, 1)
>>> f
| 1 | 0 <- tete
>>> defile(f)
0
>>> f
| 1 <- tete
```

Il arrive qu'un routeur ne puisse pas transmettre l'ensemble des paquets qu'il reçoit, par exemple si les émetteurs sont très actifs. Dans ce cas, le routeur va ignorer certains paquets.

L'une des démarches utilisées consiste à limiter la taille de la file. Si la file n'a pas atteint sa taille maximale, les paquets reçus sont enfilés. Par contre, si la taille maximale est atteinte, tous les nouveaux paquets reçus sont ignorés. Lorsqu'un paquet est transmis par le routeur, il est défilé.

Cette méthode est appelée drop tail en anglais. Un routeur suivant cet algorithme est donc paramétré avec une taille maximale de file `t_max`. Il doit garder trace, à chaque instant :

- de la file `f` contenant les paquets à transmettre ;
- de la taille actuelle `t` de cette file.

On considère la classe `Routeur_DROP_TAIL`, dont on fournit ci-dessous une partie du code.

```
class Routeur_DROP_TAIL:
    def __init__(..., ...):
        self.f = ...
        self.t_max = ...
        self.t = ...
```

9. Recopier et compléter la méthode `__init__`.

Corrigé

Le constructeur doit initialiser trois attributs :

- `self.f`, la file d'attente, initialement vide ;
- `self.t_max`, la taille maximale autorisée ;
- `self.t`, la taille actuelle de la file, initialement nulle.

La taille maximale est passée en paramètre du constructeur.

La méthode complétée est donc :

```
class Routeur_DROP_TAIL:
    def __init__(self, t_max):
        self.f = cree_file()
        self.t_max = t_max
        self.t = 0
```

La méthode `recoit` de la classe `Routeur_DROP_TAIL` prend en paramètre un paquet `p`. Cette méthode renvoie `True` si le paquet est accepté, et `False` dans le cas contraire.

On rappelle qu'un paquet est accepté si la taille de la file au moment de sa réception est strictement inférieure à la taille maximale. Dans ce cas, le paquet est enfilé.

10. Recopier et compléter la méthode `recoit` proposée ci-dessous.

```
def recoit(self, p):
    if ...:
        enfile(..., ...)
        self.t = ...
        return ...
    return ...
```

Corrigé

Un paquet est accepté si la taille actuelle de la file est strictement inférieure à la taille maximale :

$$\texttt{self.t} < \texttt{self.t\_max}.$$

Dans ce cas :

- on enfile le paquet `p` dans la file `self.f` ;
- on augmente la taille actuelle `self.t` de 1 ;
- on renvoie `True`.

Si la file est pleine, le paquet est ignoré et la méthode renvoie `False`.

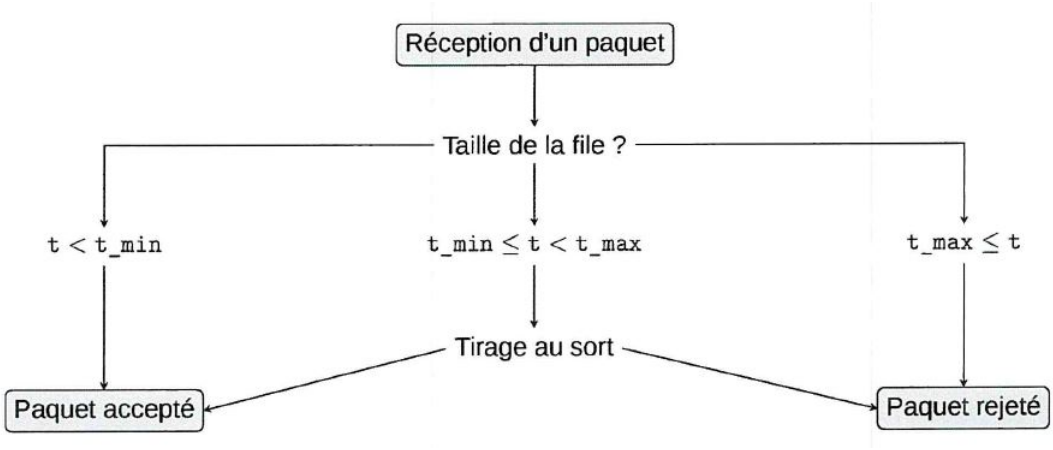
La méthode complétée est donc :

```
def recoit(self, p):
    if self.t < self.t_max:
        enfile(self.f, p)
        self.t = self.t + 1
        return True
    return False
```

La démarche décrite ci-dessus a pour désavantage d'ignorer indifféremment tous les paquets lorsque la taille maximale de la file est atteinte. Elle peut entraîner un ralentissement général des transmissions car tous les émetteurs vont ralentir leur rythme d'envoi de paquets en même temps.

Pour pallier ce problème, on se propose d'adopter la démarche suivante lors de la réception d'un paquet :

- si la taille actuelle de la file est strictement inférieure à une taille minimale `t_min`, le paquet est accepté ;
- si elle est supérieure ou égale à `t_min`, mais strictement inférieure à une taille maximale `t_max`, le paquet est rejeté aléatoirement ;
- si elle est supérieure ou égale à `t_max`, le paquet est rejeté.



On modélise cette démarche par un objet de la classe `Routeur_ALEA` possédant quatre attributs :

- la file d'attente `f`, manipulable par les fonctions décrites plus haut. Cette file est vide initialement ;
- la taille minimale `t_min`, nombre entier positif ;
- la taille maximale `t_max`, nombre entier positif ;
- la taille actuelle de la file `t`, nombre entier positif, initialement nul.

La classe `Routeur_ALEA` possède aussi une méthode `tirage_au_sort`, qui renvoie un booléen au hasard. Ainsi, l'expression :

```
self.tirage_au_sort()
```

renvoie aléatoirement `True` ou `False`.

La méthode `recoit` de la classe `Routeur_ALEA` prend en paramètre un paquet `p`. Cette méthode met en oeuvre la démarche décrite ci-dessus et renvoie `True` si le paquet est accepté, `False` dans le cas contraire.

11. Recopier et compléter la méthode `recoit` proposée ci-dessous.

```
def recoit(self, p):
    if ... < ...:
        enfile(self.f, p)
        self.t = ...
        return ...
    elif ... <= ... < ...:
        if self.tirage_au_sort():
            enfile(self.f, p)
            self.t = ...
            return ...
    return ...
```

Corrigé

On distingue les trois cas décrits dans l'énoncé.

- Si :
$$\texttt{self.t < self.t\_min},$$
alors le paquet est toujours accepté.
- Si :
$$\texttt{self.t\_min <= self.t < self.t\_max},$$
alors on effectue un tirage au sort. Si ce tirage renvoie `True`, le paquet est accepté.
- Si :
$$\texttt{self.t >= self.t\_max},$$
alors le paquet est rejeté.

Dans tous les cas où le paquet est accepté, on l'ajoute à la file et on incrémente la taille actuelle de la file.
La méthode complétée est donc :

```
def recoit(self, p):
    if self.t < self.t_min:
        enfile(self.f, p)
        self.t = self.t + 1
        return True
    elif self.t_min <= self.t < self.t_max:
        if self.tirage_au_sort():
            enfile(self.f, p)
            self.t = self.t + 1
            return True
    return False
```

Exercice 3. Bases de données, récursivité et programmation dynamique — 8 points

Cet exercice porte sur la récursivité, la programmation dynamique et les bases de données.

Les deux parties de l'exercice sont indépendantes.

Partie A : bases de données

Dans cette partie, on s'intéresse à la gestion des immeubles par une agence immobilière.
On pourra utiliser les clauses du langage SQL suivantes :

- les requêtes d'interrogation à l'aide de `SELECT`, `FROM`, `WHERE`, avec les opérateurs logiques `AND` et `OR` ;
- les jointures à l'aide de `JOIN ... ON` ;
- les requêtes d'insertion, de mise à jour et de suppression à l'aide de `INSERT`, `UPDATE` et `DELETE` ;
- les mots-clés `DISTINCT` et `ORDER BY` pour affiner les recherches.

On considère une base de données composée de deux tables : la table `immeuble` et la table `appartement`.

La table `immeuble` contient les attributs suivants :

- `id_immeuble`, qui est un numéro identifiant l'immeuble ;
- `nb_etage_immeuble`, qui est le nombre d'étages de l'immeuble ;
- `numero_immeuble`, qui est le numéro de l'immeuble dans la rue ;
- `rue_immeuble`, qui est le nom de la rue dans laquelle se trouve l'immeuble.

La table `appartement` contient les attributs suivants :

- `id_appart`, qui est un numéro identifiant l'appartement ;
- `etage_appart`, qui est l'étage où se trouve l'appartement ;
- `prix_appart`, qui est le prix de l'appartement ;
- `id_immeuble`, qui est le numéro de l'identifiant de l'immeuble dans lequel se trouve l'appartement.

Le schéma relationnel de la base de données est donné ci-dessous.

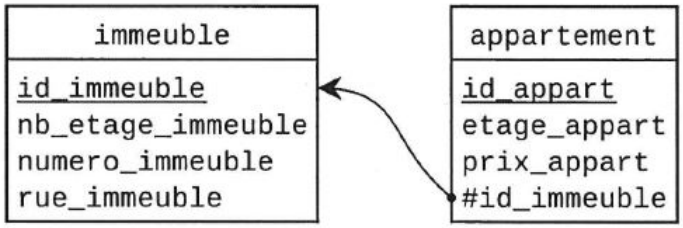


Figure 1. Schéma de la base de données

Dans ce schéma, les attributs qui forment une clé primaire sont soulignés. Les attributs qui forment une clé étrangère sont précédés d'un croisillon # avec une flèche vers l'attribut référencé.
Dans toute la suite, pour simplifier :

- l'appartement d'identifiant 603 sera appelé simplement « appartement 603 » ;
- l'immeuble d'identifiant 16 sera appelé simplement « immeuble 16 ».

1. Expliquer pourquoi le numéro d'un immeuble dans la rue n'a pas été choisi comme clé primaire de la relation `immeuble`.

Corrigé

Clé primaire

Dans une table d'une base de données relationnelle, une clé primaire est un attribut, ou un ensemble d'attributs, qui permet d'identifier de manière unique chaque enregistrement de la table.

Deux enregistrements distincts ne peuvent donc pas avoir la même valeur de clé primaire.

Le numéro d'un immeuble dans une rue ne permet pas d'identifier un immeuble de manière unique dans l'ensemble de la base.

En effet, deux immeubles situés dans deux rues différentes peuvent avoir le même numéro. Par exemple, il peut exister un immeuble au numéro 13 de la rue « Turing » et un autre immeuble au numéro 13 d'une autre rue.

Ainsi, l'attribut `numero_immeuble` seul ne garantit pas l'unicité des immeubles.

Le numéro dans la rue n'est pas une bonne clé primaire, car il n'identifie pas un immeuble de façon unique.

2. Donner une requête qui renvoie les identifiants des immeubles de la rue `'la mer'` ordonnés dans l'ordre croissant d'identifiants.

Corrigé

On veut obtenir uniquement les identifiants des immeubles. L'attribut à projeter est donc :

`id_immeuble`.

On cherche seulement les immeubles situés dans la rue `'la mer'`, donc on utilise une condition `WHERE` sur l'attribut `rue_immeuble`.

Enfin, on veut un tri dans l'ordre croissant des identifiants. On utilise donc `ORDER BY id_immeuble ASC`.

La requête est donnée ci-dessous.

```
SELECT id_immeuble
FROM immeuble
WHERE rue_immeuble = 'la mer'
ORDER BY id_immeuble ASC;
```

3. Donner une requête qui renvoie les identifiants des appartements de l'immeuble 16 et qui se situent au moins au 5^{e} étage.

Corrigé

On cherche les identifiants des appartements. L'attribut à afficher est donc :

`id_appart`.

Les appartements doivent vérifier deux conditions :

- appartenir à l'immeuble d'identifiant 16, c'est-à-dire vérifier `id_immeuble = 16;`
- se situer au moins au 5^{e} étage, c'est-à-dire vérifier `etage_appart >= 5`.

Les deux conditions doivent être vraies simultanément, on les relie donc avec `AND`.

La requête est donnée ci-dessous.

```
SELECT id_appart
FROM appartement
WHERE id_immeuble = 16
  AND etage_appart >= 5;
```

L'immeuble 16 vient d'être détruit. On souhaite effacer les données concernant cet immeuble. On commence par la requête suivante :

```
DELETE FROM immeuble WHERE id_immeuble = 16;
```

4. Expliquer pourquoi cette requête risque de rompre l'intégrité de la base de données.

Corrigé

Clé étrangère et intégrité référentielle

Une clé étrangère est un attribut d'une table qui fait référence à la clé primaire d'une autre table.

L'intégrité référentielle impose qu'une valeur de clé étrangère fasse toujours référence à une valeur existante dans la table référencée.

Dans la table `appartement`, l'attribut `id_immeuble` est une clé étrangère qui fait référence à l'attribut `id_immeuble` de la table `immeuble`.

Si on supprime directement l'enregistrement de l'immeuble 16 dans la table `immeuble`, alors il peut rester dans la table `appartement` des appartements dont l'attribut `id_immeuble` vaut 16.

Ces appartements feraient alors référence à un immeuble qui n'existe plus dans la table `immeuble`.

La requête risque de rompre l'intégrité référentielle de la base.

Il faudrait d'abord supprimer ou mettre à jour les appartements associés à l'immeuble 16, ou utiliser un mécanisme de suppression en cascade si la base le prévoit.

Suite à la construction d'un immeuble, on souhaite mettre à jour la table `immeuble`.

5. Donner une requête qui ajoute un immeuble de 6 étages, situé au numéro 13 de la rue `'Turing'`, et lui conférant l'identifiant 140.

Corrigé

On veut ajouter une nouvelle ligne dans la table `immeuble`. On utilise donc une requête `INSERT INTO`.

Les valeurs à insérer sont :

- `id_immeuble = 140;`
- `nb_etage_immeuble = 6;`
- `numero_immeuble = 13;`
- `rue_immeuble = 'Turing'.`

La requête est donnée ci-dessous.

```sql
INSERT INTO immeuble (
    id_immeuble,
    nb_etage_immeuble,
    numero_immeuble,
    rue_immeuble
)
VALUES (140, 6, 13, 'Turing');
```

Suite à la démolition d'un immeuble, l'appartement 603 a maintenant la vue sur la mer et son prix a doublé.

6. Donner une requête qui modifie en conséquence le prix de cet appartement.

Corrigé

On veut modifier une information dans une ligne existante de la table `appartement`. On utilise donc une requête `UPDATE`.

L'appartement concerné est l'appartement d'identifiant 603, donc la condition est :

$$\texttt{id\_appart} = 603.$$

Son prix doit doubler, donc on remplace `prix_appart` par :

$$2 \star \texttt{prix\_appart}.$$

La requête est donnée ci-dessous.

```sql
UPDATE appartement
SET prix_appart = 2 * prix_appart
WHERE id_appart = 603;
```

La fonction d'agrégation `MAX` permet d'obtenir la plus grande valeur d'un attribut.

Par exemple, la requête suivante renvoie le nombre maximal d'étages des immeubles dont l'identifiant est inférieur ou égal à 23 :

```sql
SELECT MAX(nb_etage_immeuble)
FROM immeuble
WHERE id_immeuble <= 23;
```

7. Donner une requête qui renvoie le prix maximal d'un appartement situé dans un immeuble de la rue `'la mer'`.

Corrigé

On cherche un prix maximal d'appartement. On utilise donc :

`MAX(prix_appart).`

Cependant, la rue de l'immeuble se trouve dans la table `immeuble`, alors que le prix de l'appartement se trouve dans la table `appartement`.
Il faut donc joindre les deux tables grâce à leur attribut commun `id_immeuble`.
La condition à appliquer est :

`rue_immeuble = 'la mer'.`

La requête est donnée ci-dessous.

```sql
SELECT MAX(prix_appart)
FROM appartement
JOIN immeuble
ON appartement.id_immeuble = immeuble.id_immeuble
WHERE rue_immeuble = 'la mer';
```

Partie B : sous-séquence strictement croissante

On considère une route perpendiculaire à la mer et des immeubles construits le long de cette route.
On dit qu'un immeuble, d'un certain nombre d'étages, bénéficie d'une vue sur la mer lorsque les immeubles situés entre lui et la mer ont moins d'étages que lui.
On souhaite que les immeubles de la rue aient tous au moins un appartement avec vue sur la mer en détruisant le moins d'immeubles possible.
Pour résoudre ce problème, on considère la liste des nombres d'étages de chaque immeuble de cette rue, en partant de la mer. Le problème revient à trouver une sous-séquence strictement croissante de longueur maximale dans cette liste.

Une sous-séquence d'une liste est une séquence d'éléments obtenue en supprimant certains éléments de la liste initiale, éventuellement aucun, sans changer l'ordre des éléments restants.
Par exemple, pour la liste :

$$L_1 = [10, 22, 9, 33, 21, 50, 41, 60],$$

les listes

$$[22, 9, 50], \qquad [10, 22] \qquad \text{et} \qquad [33]$$

sont des sous-séquences de L_1.
La longueur d'une sous-séquence est son nombre d'éléments.
On appelle sous-séquence strictement croissante d'une liste une sous-séquence telle que chaque élément est strictement supérieur au précédent.
Par exemple, pour la liste L_1 précédente :

- $[10]$ est une sous-séquence strictement croissante de longueur 1 ;
- $[9, 33]$ est une sous-séquence strictement croissante de longueur 2 ;
- $[10, 22, 33, 50, 60]$ est une sous-séquence strictement croissante de longueur maximale de L_1, et elle est de longueur 5.

Pour les questions suivantes, on définit la liste :

$$L_2 = [3, 1, 8, 2, 5].$$

8. Donner toutes les sous-séquences strictement croissantes de longueur 2 de la liste L_2.

Corrigé

On cherche toutes les sous-séquences de longueur 2 dont les deux éléments apparaissent dans le même ordre que dans L_2 et dont le second est strictement plus grand que le premier.

La liste est :

$$L_2 = [3,1,8,2,5].$$

On examine les couples possibles dans l'ordre de la liste.

- En partant de 3, on peut former :

$$[3,8] \quad \text{et} \quad [3,5].$$

- En partant de 1, on peut former :

$$[1,8], \quad [1,2] \quad \text{et} \quad [1,5].$$

- En partant de 8, on ne peut former aucune sous-séquence croissante de longueur 2, car les éléments suivants sont plus petits que 8.
- En partant de 2, on peut former :

$$[2,5].$$

Ainsi, toutes les sous-séquences strictement croissantes de longueur 2 de L_2 sont :

$$\boxed{[3,8],\ [3,5],\ [1,8],\ [1,2],\ [1,5],\ [2,5].}$$

9. Déterminer la plus longue sous-séquence strictement croissante de la liste L_2.

Corrigé

On cherche une sous-séquence strictement croissante de longueur maximale dans :

$$L_2 = [3,1,8,2,5].$$

Une sous-séquence strictement croissante de longueur 3 est :

$$[1,2,5].$$

On ne peut pas obtenir une sous-séquence strictement croissante de longueur 4, car après avoir choisi 1, les seuls éléments qui peuvent suivre dans l'ordre sont 8, 2 et 5. Le choix 8 bloque la suite, tandis que $1,2,5$ utilise déjà la chaîne croissante possible la plus longue.

Ainsi :

$$\boxed{[1,2,5] \text{ est une plus longue sous-séquence strictement croissante de } L_2.}$$

Sa longueur est :

$$\boxed{3.}$$

10. Écrire une fonction `est_strict_croissante` qui prend en paramètre une liste `seq` et renvoie `True` si la liste est strictement croissante, `False` sinon.

Corrigé

Une liste est strictement croissante si chaque élément, à partir du deuxième, est strictement supérieur à l'élément qui le précède.

On peut donc parcourir les indices de 1 à `len(seq)-1` et tester si :

`seq[i] <= seq[i-1].`

Si cette condition est vraie, alors la liste n'est pas strictement croissante et on renvoie `False`. Si aucune anomalie n'est trouvée, on renvoie `True`.

La fonction est donnée ci-dessous.

```
1  def est_strict_croissante(seq):
2      for i in range(1, len(seq)):
3          if seq[i] <= seq[i - 1]:
4              return False
5      return True
```

Récursivité

On cherche à écrire une fonction récursive qui renvoie la longueur d'une plus longue sous-séquence strictement croissante, notée `llsc`, d'un tableau donné en paramètre.

La fonction `llsc_rec` réalise ce calcul à l'aide d'une fonction auxiliaire appelée `llsc_fin`.

Le principe est le suivant :

- pour chaque élément du tableau, on considère les sous-séquences strictement croissantes qui se terminent par cet élément ;
- l'appel `llsc_fin(tab, i)` donne la longueur maximale d'une sous-séquence strictement croissante se terminant à l'indice `i` du tableau.

11. Recopier et compléter les lignes 2, 3 et 6 de la fonction `llsc_fin`.

```
1  def llsc_fin(tab, i):
2      if ...:
3          return ...
4      max_len = 1
5      for j in range(i):
6          if tab[j] < ...:
7              max_len = max(max_len, llsc_fin(tab, j) + 1)
8      return max_len
9
10 def llsc_rec(tab):
11     n = len(tab)
12     return max([llsc_fin(tab, i) for i in range(n)])
```

Corrigé

La fonction `llsc_fin(tab, i)` renvoie la longueur d'une plus longue sous-séquence strictement croissante se terminant à l'indice `i`.

- Si `i == 0`, la sous-séquence se terminant au premier élément ne peut contenir que cet élément. Sa longueur est donc 1.

- Pour construire une sous-séquence strictement croissante qui se termine à l'indice `i`, on peut la faire précéder d'une sous-séquence se terminant à un indice `j < i`, à condition que :

$$\texttt{tab[j] < tab[i]}.$$

La fonction complétée est donnée ci-dessous.

```
1  def llsc_fin(tab, i):
2      if i == 0:
3          return 1
4      max_len = 1
5      for j in range(i):
6          if tab[j] < tab[i]:
7              max_len = max(max_len, llsc_fin(tab, j) + 1)
8      return max_len
9
10 def llsc_rec(tab):
11     n = len(tab)
12     return max([llsc_fin(tab, i) for i in range(n)])
```

Programmation dynamique

Afin de résoudre le même problème, on utilise maintenant la programmation dynamique. On cherche à écrire une fonction `llsc_dyn`.
Le principe est le suivant :

- on crée une liste `dyn` telle que `dyn[i]` contient la longueur d'une plus longue sous-séquence strictement croissante se terminant à l'indice `i` ;
- pour chaque indice `i`, on parcourt les indices `j < i` ;
- si `tab[j] < tab[i]`, alors on peut prolonger une sous-séquence se terminant à l'indice `j` avec l'élément d'indice `i`.

12. Recopier et compléter les lignes 7 et 8 de la fonction `llsc_dyn`. On pourra utiliser la fonction `max`, qui renvoie le maximum des valeurs données en paramètres.

```
1  def llsc_dyn(tab):
2      n = len(tab)
3      dyn = [1] * n
4      for i in range(1, n):
5          for j in range(i):
6              if tab[j] < tab[i]:
7                  dyn[i] = max(..., ...)
8      return ...
```

Corrigé

La liste `dyn` est initialisée avec des 1, car chaque élément du tableau peut former à lui seul une sous-séquence strictement croissante de longueur 1.

Lorsque `tab[j] < tab[i]`, on peut prolonger une sous-séquence strictement croissante se terminant en `j` en ajoutant `tab[i]`. La longueur obtenue est alors :

$$\texttt{dyn[j] + 1}.$$

On conserve la meilleure valeur entre l'ancienne valeur de `dyn[i]` et cette nouvelle possibilité.

À la fin, la longueur d'une plus longue sous-séquence strictement croissante du tableau est le maximum des valeurs de `dyn`.

La fonction complétée est donnée ci-dessous.

```
def llsc_dyn(tab):
    n = len(tab)
    dyn = [1] * n
    for i in range(1, n):
        for j in range(i):
            if tab[j] < tab[i]:
                dyn[i] = max(dyn[i], dyn[j] + 1)
    return max(dyn)
```

13. Citer un avantage de cette implémentation par rapport à l'implémentation récursive.

Corrigé

L'implémentation récursive recalcule plusieurs fois les mêmes valeurs. Par exemple, la longueur d'une plus longue sous-séquence strictement croissante se terminant à un indice donné peut être redemandée dans plusieurs appels récursifs.

La programmation dynamique évite ces recalculs : chaque valeur `dyn[i]` est calculée une fois, puis mémorisée dans la liste `dyn`.

Ainsi, l'implémentation dynamique est beaucoup plus efficace en temps de calcul.

La programmation dynamique évite les recalculs et rend l'algorithme plus rapide.

Fin du devoir