BACCALAURÉAT GÉNÉRAL

ÉPREUVE D'ENSEIGNEMENT DE SPÉCIALITÉ

SESSION 2026

NUMÉRIQUE ET SCIENCES INFORMATIQUES

JOUR 1

Durée de l'épreuve : **3 heures 30**

L'usage de la calculatrice n'est pas autorisé.

Dès que ce sujet vous est remis, assurez-vous qu'il est complet.

Ce sujet comporte 16 pages numérotées de 1/16 à 16/16.

Le sujet est composé de trois exercices indépendants.

Le candidat traite les trois exercices.

Exercice 1 (6 points)

Cet exercice porte sur la programmation orientée objet et la récursivité.

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. Lorsque c'est à son tour de jouer, 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, en diagonale). Le premier à y parvenir a gagné.

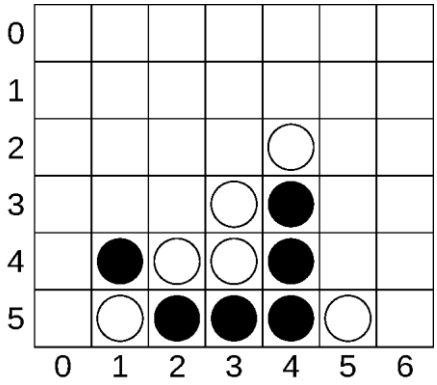


Figure 1. Une partie gagnée par le joueur blanc

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 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.

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.

```
1 def joue(self, colonne, joueur):
2     ligne = 5  # on part de la rangée la plus basse
3     while ligne!=-1 and self.grille[ligne][colonne]!=0:
4         ligne = ...
5     if ligne != -1:
6         self.grille[ligne][colonne] = ...
7         return ...
8     else:
9         return ...
```

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`.

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.

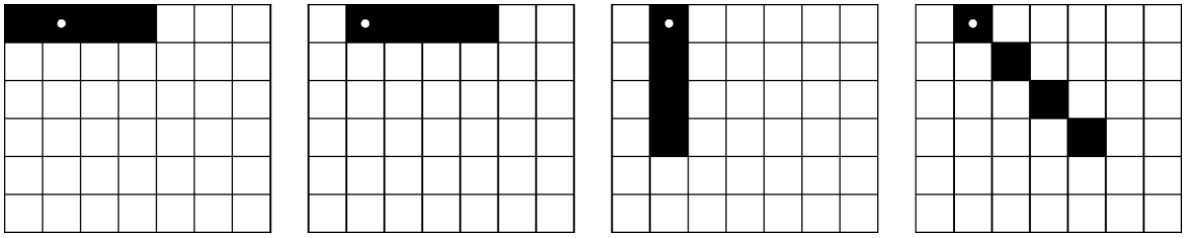


Figure 2. Alignements contenant la case (0, 1).

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.

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

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 $-\big(100 + 10 \times (niveau_max - niveau)\big)$ pour le joueur 1 et $\big(100 + 10 \times (niveau_max - niveau)\big)$ 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.

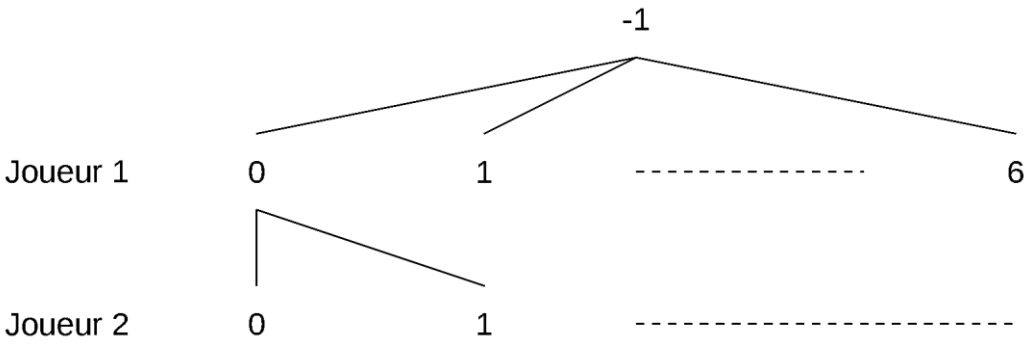


Figure 3. Schéma partiel de l'arbre de coups avec niveau_max = 2 où l'on a indiqué la valeur de l'attribut colonne

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.

```
1 class Noeud:
2     def __init__(self, colonne):
3         ...
4         ...
5         ...
```

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.

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 = ...
```

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`.

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.

Exercice 2 (6 points)

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

Voici un schéma du réseau de l'entreprise Gamerzz, qui propose à ses clients :

- une salle pour jouer à des jeux vidéos en ligne (Salle Gaming Online) ;
- une autre pour jouer en réalité virtuelle (Salle Gaming VR) ;
- un site pour réserver, organiser des évènements et gérer les classements des joueurs en réalité virtuelle.

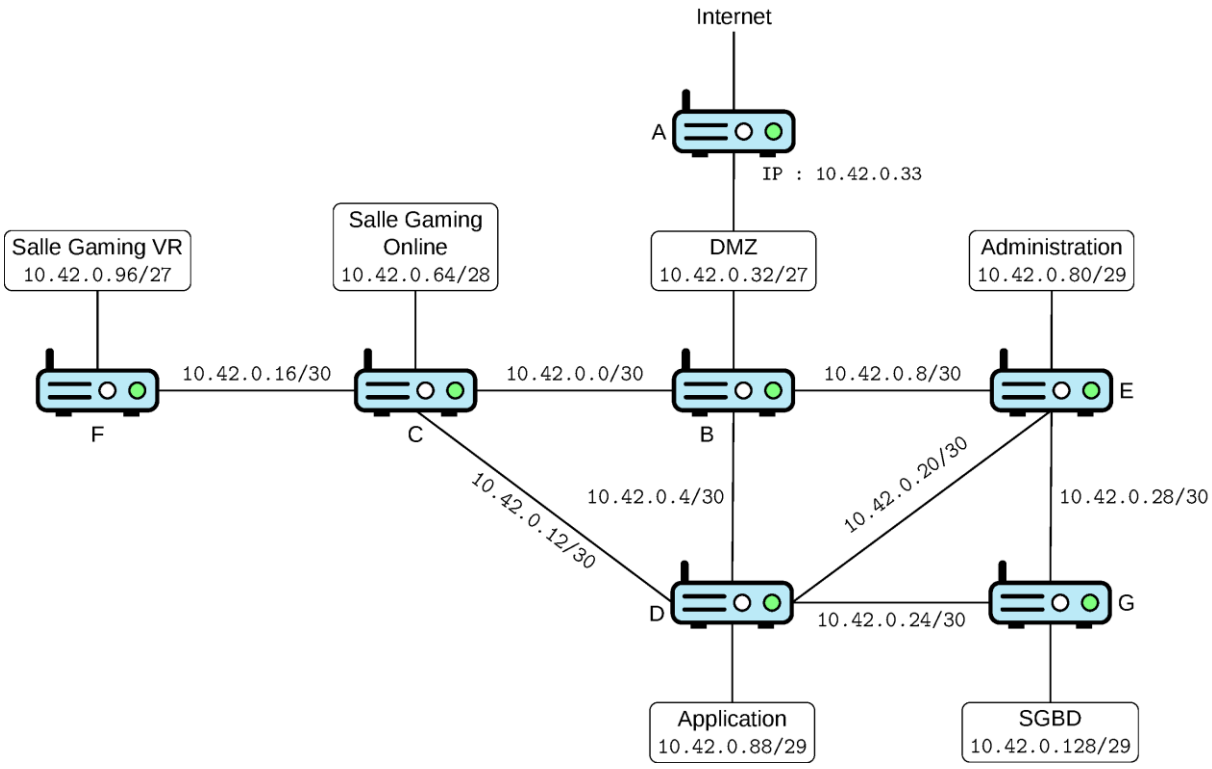


Figure 1. Réseau de l'entreprise Gamerzz

On y a fait figurer les différents sous-réseaux qui la composent en donnant leur notation CIDR.

La notation `a.b.c.d/n`, appelée notation CIDR (Classless Inter Domain Routing), signifie que les n premiers bits à gauche de l'adresse IP représentent la partie « réseau », les bits à droite qui suivent 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.

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

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

Les réseaux qui interconnectent les routeurs sont en /30 ce qui permet d'attribuer une adresse à exactement deux machines, les deux autres adresses étant l'adresse de 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.

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 alors 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.

On indique en figure 2 les débits des liaisons entre routeurs.

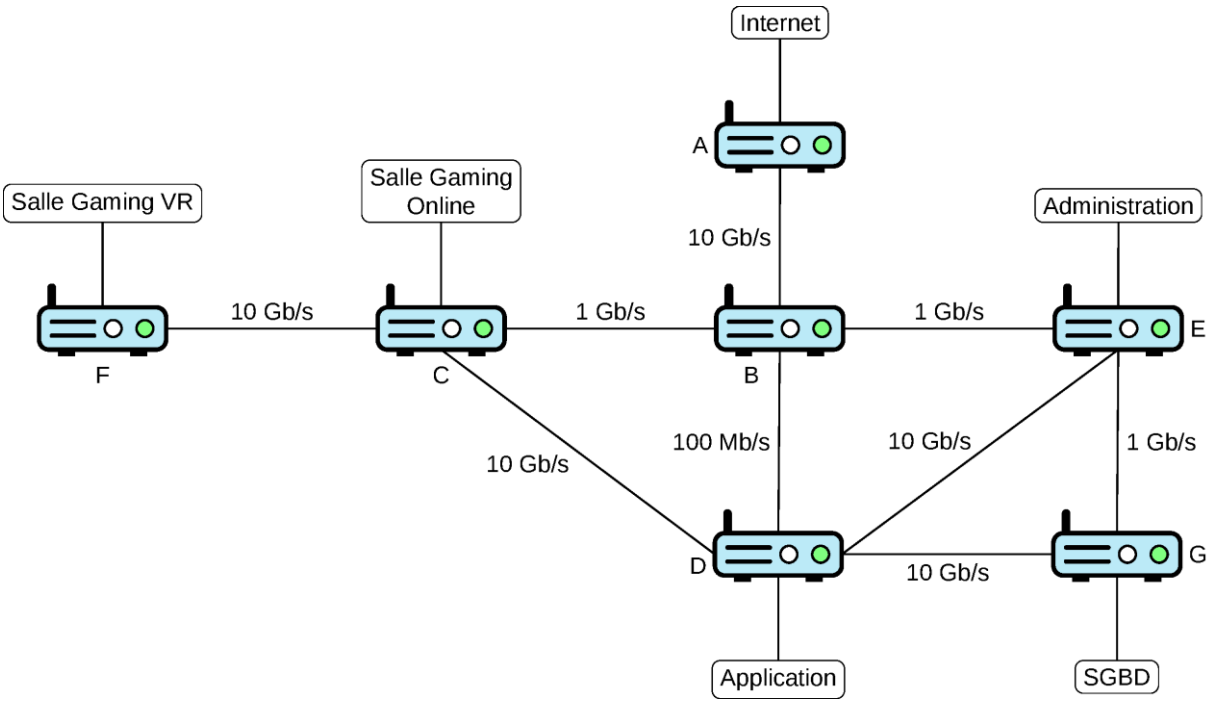


Figure 2. Débits des liaisons

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.

6. Une information venue d'un client extérieur, par exemple depuis Internet, est acheminée vers le réseau « Application » pour être traitée. Donner un des ordres possibles des routeurs par lesquels cette information transite à partir du routeur A en suivant le protocole OSPF.

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.

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.

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` et ajoute `x` dans `f` ;
- `defile` : prend en paramètre une file `f` non vide et supprime l'élément de `f` le plus anciennement ajouté et renvoie sa valeur.

On a par exemple :

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

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` et il doit garder trace, à chaque instant, de la file `f` contenant les paquets à transmettre et de la taille `t` de celle-ci.

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

```
1 class Routeur_DROP_TAIL:
2     def __init__(..., ...):
3         self.f = ...
4         self.t_max = ...
5         self.t = ...
```

9. Recopier et compléter la méthode `__init__`.

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

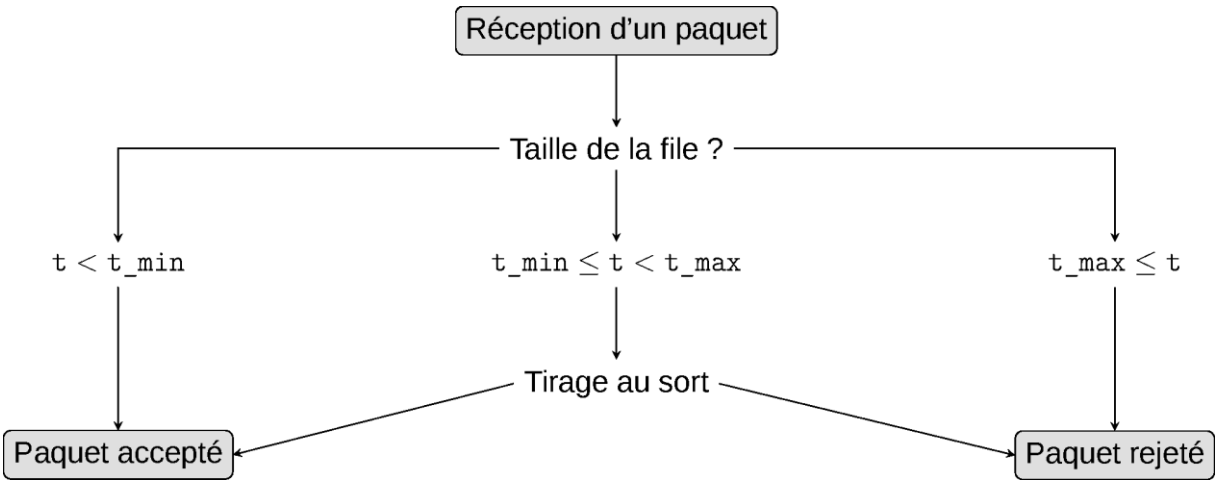
10. Recopier et compléter la méthode `recoit` proposée ci-dessous :

```
1     def recoit(self, p):
2         if ...:
3             enfile(..., ...)
4             self.t = ...
5             return ...
6         return ...
```

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 palier 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()` renverra aléatoirement `True` ou `False`.

La méthode `recoit` de la classe `Routeur_ALEA` prend en paramètre un paquet `p`. Cette méthode met en œuvre 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 ...
```

Exercice 3 (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

Dans cette partie, on s'intéresse à la gestion des immeubles par une agence immobilière.

On pourra utiliser les clauses du langage SQL pour :

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

On considère une base de données composée des deux tables suivantes :

- `immeuble` :
  - `id_immeuble` est un numéro identifiant l'immeuble ;
  - `nb_etage_immeuble` est le nombre d'étages de l'immeuble ;
  - `numero_immeuble` est le numéro de l'immeuble dans la rue ;
  - `rue_immeuble` est le nom de la rue dans laquelle se trouve l'immeuble.
- `appartement` :
  - `id_appart` est un numéro identifiant l'appartement ;
  - `etage_appart` est l'étage où se trouve l'appartement ;
  - `prix_appart` est le prix de l'appartement ;
  - `id_immeuble` 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 en figure 1, avec la convention que les attributs formant une clé primaire sont soulignés tandis que ceux d'une clé étrangère sont précédés d'un croisillon (symbole #) avec une flèche vers l'attribut référencé.

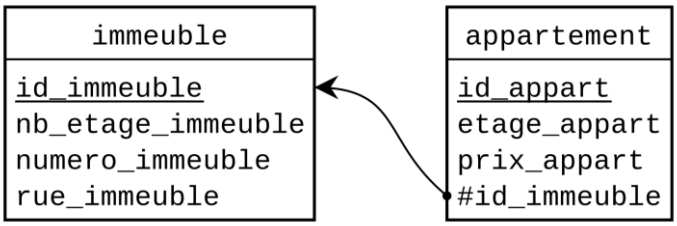


Figure 1. Schéma de la base de données

Dans toute la suite, pour simplifier, l'appartement d'identifiant 603 sera désigné simplement par "appartement 603" et l'immeuble d'identifiant 16 sera désigné par "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`.

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

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

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.

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'` en lui conférant l'identifiant 140.

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.

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 :

```
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'`.

Partie B

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 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 : il s'agit de trouver ce qu'on appelle une *sous-séquence strictement croissante de longueur maximale de cette liste*.

Une *sous-séquence* d'une liste est une suite d'éléments obtenue en supprimant certains éléments, éventuellement aucun, de la liste d'origine sans changer l'ordre des éléments restants.

Exemples : pour la liste `L1 = [10, 22, 9, 33, 21, 50, 41, 60]`, les listes `[22, 9, 50]`, `[10, 22]` ou `[33]` sont des sous-séquences.

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.

Exemples : pour la liste `L1` 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 `L1` ; elle est de longueur 5.

Pour les questions suivantes, on définit la liste `L2 = [3, 1, 8, 2, 5]`.

8. Donner toutes les sous-séquences strictement croissantes de longueur 2 de la liste `L2`.

9. Déterminer la plus longue sous-séquence strictement croissante de la liste `L2`.

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.

Récursivité

On cherche à écrire une fonction récursive qui renvoie la longueur d'une plus longue sous-séquence strictement croissante (`llsc`) d'un tableau donné en paramètre. La fonction `llsc_rec` réalise ce calcul à l'aide d'une fonction auxiliaire appelée `llsc_fin`.

Principe :

- pour chaque élément du tableau, on considère deux possibilités : l'inclure ou non dans la sous-séquence ;
- l'appel de la fonction auxiliaire `llsc_fin(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)])
```

Programmation dynamique

Afin de résoudre le même problème, cette fois-ci on utilise la programmation dynamique. On cherche à écrire la fonction `llsc_dyn`.

Principe :

- 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 `i`, on parcourt les indices `j < i` et si `tab[j] < tab[i]`, on met à jour `dyn[i]`.

12. Recopier et compléter les lignes 7 et 8 de la fonction `llsc_dyn`. On pourra utiliser la fonction native de Python `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 ...
```

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