Numérique et sciences informatiques

Baccalauréat général 2026 — épreuve de spécialité

Métropole, session 2026 (26-NSIJ1ME1)

Corrigé

Trois exercices

Ibrahim Alame — Adloun, édition 2026

Exercice 1 (6 points)

Sujet — Exercice 1 — 6 points

Cet exercice porte sur les réseaux, les protocoles de routage et la sécurisation des communications.

Le réseau informatique d'un lycée est réparti sur plusieurs sites : campus, internat, services administratifs, etc.

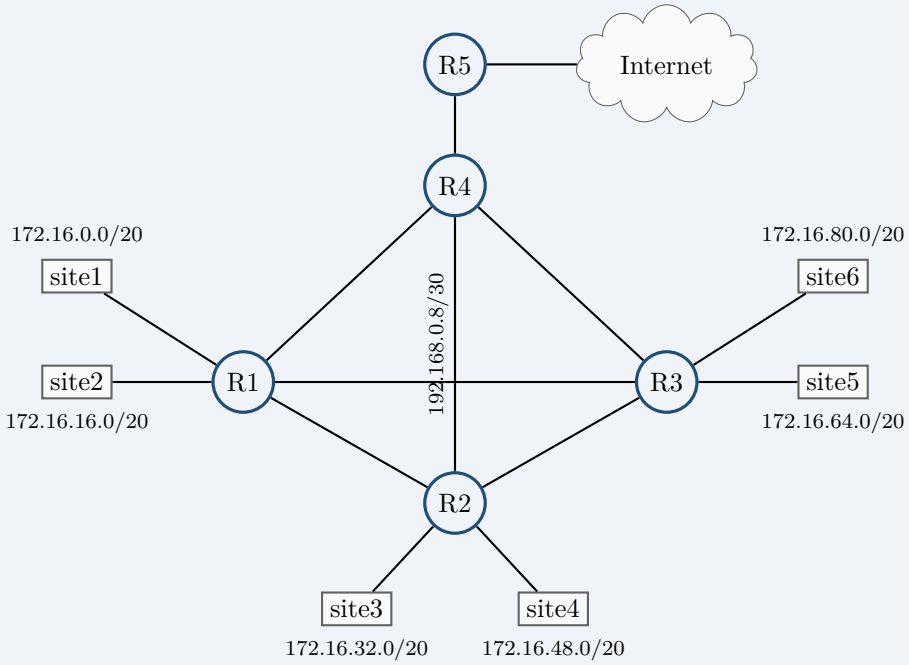

Figure 1. Schéma du réseau du lycée

Chaque site possède :

- un sous-réseau dédié, avec un adressage IP propre ;
- un routeur permettant la communication entre sites et vers Internet.

On utilise la notation CIDR pour définir l'adressage d'un réseau : la notation `a.b.c.d/n` signifie que les `n` bits de poids fort (à gauche de l'adresse IP) désignent la partie réseau de l'adresse IP et que les bits suivants de poids faible (à droite de l'adresse IP) désignent la partie machine. Dans un même sous-réseau, toutes les machines utilisent des adresses IP ayant la même partie réseau.

L'administrateur réseau a établi la convention d'adressage suivante :

- la passerelle (*gateway*) de chaque site correspond à la dernière adresse IP disponible du sous-réseau ;
- elle est affectée à l'interface du routeur connecté au site.

Exemple :
Pour le réseau du site6 (`172.16.80.0/20`) :

- Adresse réseau : `172.16.80.0`
- Adresse de passerelle : `172.16.95.254`
- Adresse de l'interface du routeur `R3` dans le site6 : `172.16.95.254`

Partie A : Configuration IP du site3 Un hôte situé dans le site3 possède la configuration suivante dans son fichier `/etc/network/interfaces` :

```
iface eth0 inet static
   address 172.16.32.15
   netmask 255.255.240.0
   gateway 172.16.48.254
```

1. Convertir les adresses suivantes en binaire sur 32 bits :

- Adresse IP : `172.16.32.15`
- Masque de sous-réseau : `255.255.240.0`

Donner le résultat sous la forme :

```
Adresse IP : XXXXXXXX.XXXXXXXX.XXXXXXXX.XXXXXXXX
Masque     : XXXXXXXX.XXXXXXXX.XXXXXXXX.XXXXXXXX
```

2. Indiquer le nombre d'hôtes que le réseau peut accueillir, ainsi que la première et la dernière adresse IP utilisables par ces hôtes.
L'administrateur a remarqué que depuis cet hôte, il pouvait accéder aux serveurs ou imprimantes du site3 mais ne pouvait pas accéder à Internet.
Il effectue un test de connectivité vers l'adresse `172.16.47.254` : ce test réussit.
3. Identifier dans la configuration de cet hôte le paramètre mal configuré et proposer une correction.
Le lien entre les routeurs R2 et R4 est configuré avec des adresses appartenant au réseau `192.168.0.8/30`.
4. Proposer deux adresses IP à attribuer aux interfaces des routeurs R2 et R4 pour établir ce lien.

Partie B On s'intéresse ici au routage dynamique appliqué dans le réseau du lycée.
L'administrateur lance la commande suivante dans un terminal depuis un hôte du site3. Il observe le résultat :

```
$ traceroute education.gouv.fr

Détermination de l'itinéraire vers education.gouv.fr
  1     4 ms     7 ms     9 ms  R2 [172.16.47.254]
  2    22 ms     6 ms     8 ms  R3 [192.168.0.14]
  3    22 ms     7 ms     9 ms  R4 [192.168.0.6]
  4    21 ms     6 ms     8 ms  R5 [192.168.0.1]
  5    19 ms     7 ms     9 ms  internet-router [...]
  6     ...
```

Le protocole de routage dynamique activé sur les routeurs est RIP. On rappelle que dans le protocole RIP, le coût d'une route est donnée par le nombre de liens réseaux empruntés sur cette route.

5. D'après le résultat de la commande passée par l'administrateur, donner le chemin suivi par l'information et son coût pour aller de R2 à R5.
6. Expliquer en quoi le réseau présente un dysfonctionnement et formuler une cause possible de ce problème.
Le protocole de routage dynamique activé sur les routeurs est maintenant OSPF.
On rappelle la relation suivante entre le débit et le coût d'une liaison entre deux routeurs :

$$cout = \frac{10^8}{debit}$$

7. Sur un réseau informatique, pour chacun des deux termes « débit » et « coût » d'une liaison, préciser si l'on souhaite le minimiser ou le maximiser.
On donne les informations suivantes sur les types de liaisons utilisées entre les routeurs :

- Liaison R1-R2 : Fast Ethernet
- Liaison R1-R3 : Fibre optique
- Liaison R1-R4 : Fast Ethernet
- Liaison R2-R3 : Ethernet
- Liaison R2-R4 : Ethernet
- Liaison R3-R4 : Fibre optique
- Liaison R4-R5 : Fibre optique

On donne également les débits des différentes liaisons utilisées sur le réseau :

- Ethernet : 10 Mbits/s
- Fast Ethernet : 100 Mbit/s
- Fibre optique : 4 Gbit/s

8. Donner, en précisant son coût, le chemin emprunté par un paquet de données depuis le site4 jusqu'à Internet.

Partie C On s'intéresse maintenant à la sécurisation des échanges de données entre les ordinateurs du réseau du lycée.
Un élève de l'internat se situant dans le site2 a ouvert un logiciel malveillant ayant infecté son ordinateur. Le pirate a maintenant pris le contrôle de son ordinateur et peut donc maintenant accéder au réseau du lycée.
Alice se trouve dans le site4 et Bob dans le site5, et entament une conversation privée en utilisant le réseau du lycée. On suppose que l'échange débute après l'arrivée du pirate sur le réseau.
9. Entre le chiffrement symétrique ou asymétrique, donner en justifiant lequel des deux Alice et Bob doivent préférer utiliser pour éviter que le pirate ne puisse connaître le contenu de leurs échanges.
10. Expliquer en quoi le protocole HTTPS est plus sécurisé que le protocole HTTP pour les échanges entre Alice et Bob.

Corrigé

Partie A : Configuration IP du site3 **1.** Une adresse IPv4 est codée sur 32 bits, soit quatre octets ; on convertit chaque octet séparément en l'écrivant comme somme de puissances de 2 $(128, 64, 32, 16, 8, 4, 2, 1)$:

- $172 = 128 + 32 + 8 + 4 \rightarrow$ `10101100` ;

— $16 \to$ `00010000` ;
— $32 \to$ `00100000` ;
— $15 = 8+4+2+1 \to$ `00001111` ;
— $255 = 128+64+32+16+8+4+2+1 \to$ `11111111` ;
— $240 = 128+64+32+16 \to$ `11110000` ;
— $0 \to$ `00000000`.

```
Adresse IP : 10101100.00010000.00100000.00001111
Masque     : 11111111.11111111.11110000.00000000
```

2. Le masque comporte $8+8+4 = 20$ bits à 1 : c'est le réseau `172.16.32.0/20`, ce qui laisse $32-20 = 12$ bits pour la partie machine, donc $2^{12} = 4\,096$ adresses. Deux d'entre elles sont réservées : l'adresse du réseau (tous les bits machine à 0) et l'adresse de diffusion (*broadcast*, tous les bits machine à 1). Le réseau peut donc accueillir $2^{12} - 2 = 4\,094$ hôtes.
L'adresse du réseau s'obtient par un ET logique bit à bit entre l'adresse et le masque : sur le troisième octet, `00100000` $\wedge$ `11110000` = `00100000` = 32, d'où `172.16.32.0`. L'adresse de diffusion met à 1 les 12 bits machine : troisième octet `00101111` = 47, quatrième 255, soit `172.16.47.255`. Les adresses utilisables par les hôtes vont donc de `172.16.32.1` (première) à `172.16.47.254` (dernière).
Ce que le correcteur attend : ne pas oublier de retirer les deux adresses réservées (4 094 et non 4 096), et ne pas s'arrêter à `172.16.32.255` : le bloc `/20` couvre les troisièmes octets de 32 à 47. On peut ajouter que, par la convention de l'administrateur, la dernière adresse `172.16.47.254` est occupée par la passerelle, ce qui laisse 4 093 adresses pour les autres machines.
3. Le paramètre mal configuré est la passerelle (`gateway 172.16.48.254`). Cette adresse n'appartient pas au sous-réseau de l'hôte : le ET logique avec le masque donne 48 = `00110000`, soit le réseau `172.16.48.0/20`, celui du site4, et non `172.16.32.0/20`. Les symptômes sont cohérents : pour joindre une machine du même sous-réseau (serveurs, imprimantes du site3) l'hôte n'a pas besoin de passerelle, cela fonctionne ; pour joindre Internet, il doit remettre ses paquets à la passerelle, qui est inaccessible puisqu'elle n'est pas sur son réseau local.
Le test de connectivité vers `172.16.47.254` réussit : c'est la dernière adresse utilisable du sous-réseau du site3, donc, selon la convention, l'adresse de l'interface du routeur R2 dans le site3. Correction : `gateway 172.16.47.254`.
4. Dans le réseau `192.168.0.8/30`, il reste $32-30 = 2$ bits pour la partie machine, soit $2^2 = 4$ adresses : `192.168.0.8` (adresse du réseau), `192.168.0.9`, `192.168.0.10` et `192.168.0.11` (adresse de diffusion). Seules deux adresses sont utilisables, ce qui suffit exactement pour un lien point à point entre deux routeurs : par exemple `192.168.0.9` pour l'interface de R2 et `192.168.0.10` pour celle de R4 (ou l'inverse).

Partie B **5.** Le `traceroute` liste, dans l'ordre, les routeurs traversés depuis l'hôte du site3 : R2 (sa passerelle, `172.16.47.254`), puis R3, puis R4, puis R5, puis le routeur du fournisseur d'accès. Le chemin suivi de R2 à R5 est donc

$$\text{R2} \to \text{R3} \to \text{R4} \to \text{R5},$$

qui emprunte trois liens réseau : son coût RIP est 3.
6. RIP choisit toujours la route de plus petit nombre de sauts. Or le schéma montre un lien direct entre R2 et R4 (le réseau `192.168.0.8/30`) : la route R2 $\to$ R4 $\to$ R5 n'emprunte que deux liens, coût 2, strictement inférieur au coût 3 de la route observée. Le réseau ne route donc pas les paquets par le meilleur chemin : c'est le dysfonctionnement.

Cause possible : le lien R2–R4 est hors service ou n'est pas encore opérationnel (câble débranché, interface désactivée, adresses du réseau `192.168.0.8/30` non encore configurées sur les deux routeurs, comme le suggère la question 4). Faute de ce lien, RIP a convergé vers la meilleure route restante, qui passe par R3. Une autre cause recevable : les tables de routage RIP ne sont pas encore mises à jour après une modification du réseau (RIP échange ses tables toutes les 30 secondes et converge lentement).

7. Le débit est la quantité de données transmises par seconde : on souhaite le maximiser. Le coût d'une liaison, inversement proportionnel au débit d'après la formule $cout = 10^8/debit$, mesure ce que « dépense » un paquet en empruntant la liaison : on souhaite le minimiser. OSPF choisit précisément la route de coût total minimal, c'est-à-dire de débit maximal.

8. On calcule d'abord le coût de chaque type de liaison :

— Ethernet, 10 Mbit/s $= 10^7$ bit/s : $cout = 10^8/10^7 = 10$;

— Fast Ethernet, 100 Mbit/s $= 10^8$ bit/s : $cout = 10^8/10^8 = 1$;

— Fibre optique, 4 Gbit/s $= 4 \times 10^9$ bit/s : $cout = 10^8/(4 \times 10^9) = 1/40 = 0{,}025$.

Le site4 est relié au routeur R2, et seul R5 est relié à Internet : il faut donc trouver le chemin de coût minimal de R2 à R5. On compare les chemins possibles :

Chemin de R2 à R5	**Liaisons empruntées**	**Coût**
R2 – R4 – R5	Ethernet, fibre	$10 + 0{,}025 = 10{,}025$
R2 – R3 – R4 – R5	Ethernet, fibre, fibre	10,05
R2 – R1 – R4 – R5	Fast Ethernet, Fast Ethernet, fibre	2,025
R2 – R1 – R3 – R4 – R5	Fast Ethernet, fibre, fibre, fibre	$1 + 3 \times 0{,}025 = 1{,}075$
R2 – R3 – R1 – R4 – R5	Ethernet, fibre, Fast Ethernet, fibre	11,05

Le chemin de coût minimal est

$$\text{site4} \to \text{R2} \to \text{R1} \to \text{R3} \to \text{R4} \to \text{R5} \to \text{Internet},$$

de coût $1 + 0{,}025 + 0{,}025 + 0{,}025 = 1{,}075$.

Ce que le correcteur attend : avec OSPF, le chemin le moins coûteux n'est pas le plus court en nombre de sauts : quatre liens, contre deux pour R2 – R4 – R5, mais ce dernier emprunte un lien Ethernet à 10 Mbit/s dont le coût (10) écrase tout le reste. C'est la différence de fond entre RIP (nombre de sauts) et OSPF (débit des liens). Le coût de la fibre doit être calculé, 0,025, et non arrondi à 0 : c'est lui qui départage R1 – R3 – R4 (coût 0,05) de R1 – R4 (coût 1).

Remarque : dans les routeurs réels, le coût OSPF est un entier et tout résultat inférieur à 1 est ramené à 1 ; avec cette convention la fibre coûterait 1 et le chemin retenu serait R2 – R1 – R4 – R5 (coût $1 + 1 + 1 = 3$, contre 4 pour R2 – R1 – R3 – R4 – R5). Le sujet impose la formule $cout = 10^8/debit$ sans plancher : on l'applique telle quelle, et la réponse attendue est bien le chemin par R1, R3 et R4, de coût 1,075. Un candidat qui mentionne la convention du plancher, puis applique la formule du sujet, ne peut pas être pénalisé.

Partie C **9.** Alice et Bob doivent préférer le chiffrement asymétrique.

Avec un chiffrement symétrique, une même clé secrète sert à chiffrer et à déchiffrer ; Alice et Bob devraient donc se transmettre cette clé avant leur premier message, à travers le réseau du lycée. Or le pirate est déjà présent sur ce réseau quand l'échange débute : il peut intercepter la clé au moment de sa transmission, puis déchiffrer toute la conversation.

Avec un chiffrement asymétrique, chacun possède une paire de clés : une clé publique, qui peut circuler en clair, et une clé privée, qui ne quitte jamais son ordinateur. Alice chiffre ses messages avec la clé publique de Bob ; seul Bob, avec sa clé privée, peut les déchiffrer, et réciproquement.

Le pirate peut bien capturer les clés publiques et les messages chiffrés, il ne possède aucune clé privée : il ne peut pas lire les échanges. Rien de secret ne transite jamais sur le réseau.
En pratique, comme l'asymétrique est lent, on l'utilise pour échanger de façon sûre une clé de session symétrique, puis on chiffre la conversation avec celle-ci (chiffrement hybride) ; mais la réponse attendue au « lequel des deux » est bien l'asymétrique, seul capable de résoudre le problème de l'échange de clé en présence d'un espion.
10. Avec HTTP, les requêtes et les réponses circulent en clair sur le réseau : le pirate, qui contrôle une machine du site2 et a accès au réseau du lycée, peut capturer les paquets et lire directement le contenu de la conversation d'Alice et Bob, voire le modifier.
HTTPS est HTTP encapsulé dans le protocole TLS (port 443 au lieu de 80). Il apporte trois garanties absentes de HTTP :

— la *confidentialité* : les données sont chiffrées de bout en bout, selon le principe hybride de la question 9 (la clé de session symétrique est transmise chiffrée avec la clé publique du serveur) ; un paquet intercepté est illisible ;

— l'*authentification* : le serveur présente un certificat signé par une autorité de confiance, qui prouve son identité ; le pirate ne peut pas se faire passer pour Bob (attaque de « l'homme du milieu ») ;

— l'*intégrité* : toute altération des données en transit est détectée.

Avec HTTPS, le pirate voit passer des paquets entre Alice et Bob, mais n'en connaît ni le contenu ni ne peut les falsifier sans être détecté.

Exercice 2 (6 points)

Sujet — Exercice 2 — 6 points

Cet exercice porte sur le binaire, les graphes, la récursivité.
Le jeu du Recto-Verso, de la famille des solitaires, se compose de neuf jetons disposés sur un plateau carré. Le plateau a 3 lignes et 3 colonnes. Chaque jeton possède une face Recto, dessinée en couleur noire sur les figures, et une face Verso, dessinée en blanc.

Partie A : représentation binaire des configurations On appelle configuration l'état du plateau. La figure 1 illustre un exemple de configuration.

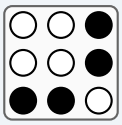

Figure 1. Illustration d'une configuration

1. Justifier que le nombre total de configurations possibles est 2^9.
Le but du jeu est, depuis une configuration quelconque, de retourner tous les jetons sur leur face Recto (noire). Pour ce faire, le seul type d'opération autorisé est de choisir une ligne ou une colonne ou une diagonale, et d'en retourner tous les jetons. On appelle une telle opération un coup. La figure 2 illustre les huit différents coups possibles à partir d'une configuration donnée.

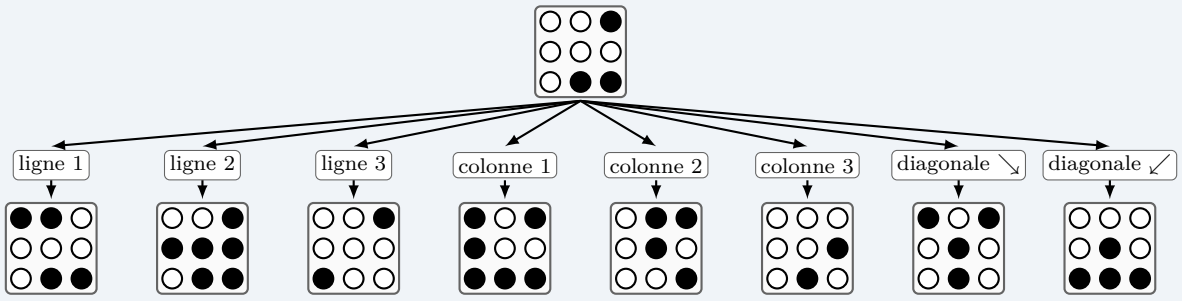

Figure 2. Les coups possibles depuis la configuration `67`

2. Indiquer en justifiant si la séquence de coups suivante, jouée depuis la configuration donnée en figure 1, permet de mener à la victoire : ligne 1, ligne 2, colonne 2.
Une configuration est représentée, ligne par ligne et de haut en bas, par une liste de bits en considérant les jetons Recto comme des `1` et les jetons Verso comme des `0`. Ainsi on lit la configuration initiale de la figure 2 comme la liste `[0, 0, 1, 0, 0, 0, 0, 1, 1]`.
On peut aussi lire la liste de bits d'une configuration comme l'écriture binaire d'un nombre entier. Pour l'exemple de la figure 2, la configuration initiale est donc aussi représentée par l'entier `67`.
3. En détaillant le calcul, donner l'entier représentant la configuration illustrée en figure 1.

4. Implémenter une fonction `representant` prenant en paramètre une liste de `0` et de `1` et renvoyant l'entier correspondant.
Dans la suite on considère à disposition une fonction `binaire` prenant réciproquement en paramètre un entier et renvoyant son écriture en binaire sous forme d'une liste de neuf `0` ou `1`.
L'opérateur ou exclusif, noté $\oplus$, a pour table de vérité :

X	Y	X⊕Y
0	0	0
0	1	1
1	0	1
1	1	0

La fonction `xor` définie ci-dessous prend en paramètres deux bits `b1`, `b2` (de type `int`) et renvoie le résultat de `b1`⊕`b2`.

```
def xor(b1, b2):
    if b1 == b2:
        return 0
    else:
        return 1
```

L'opérateur $\oplus$ peut être appliqué sur deux listes de même longueur `l_x` et `l_y` contenant plusieurs bits : pour chaque indice de ces listes, on applique le $\oplus$ aux deux bits positionnés à cet indice. Par exemple, si `l_x = [1, 1]` et `l_y = [0, 1]`, alors `l_x`⊕`l_y` vaut `[1, 0]`, puisque $1 \oplus 0$ vaut 1, et $1 \oplus 1$ vaut 0.
5. Implémenter une fonction `xor_etendu` qui prend en paramètre deux listes de bits de même longueur `l_x` et `l_y` et renvoie la liste `l_x`⊕`l_y`.
On peut observer que pour un bit X donné, l'opération X⊕1 a pour résultat l'inverse de X et l'opération X⊕0 a pour résultat X.
Si X représente un jeton, alors X⊕1 le retourne et X⊕0 le laisse inchangé.
Appliquer un coup à une configuration revient à lister les jetons retournés et les jetons non-retournés. On peut donc représenter chaque coup par une liste de bits. Par exemple le coup « ligne 1 » est représenté par la liste `[1, 1, 1, 0, 0, 0, 0, 0, 0]` et l'appliquer à la configuration `67` revient à calculer `[0, 0, 1, 0, 0, 0, 0, 1, 1]`⊕`[1, 1, 1, 0, 0, 0, 0, 0, 0]`.
La figure 3 illustre les 8 coups possibles de la même manière au moyen du $\oplus$ (par souci de lisibilité la notation en liste y est simplifiée) :

Coup	67	**masque**	**résultat**	**entier**
ligne 1	001000011	⊕ 111000000	110000011	387
ligne 2	001000011	⊕ 000111000	001111011	123
ligne 3	001000011	⊕ 000000111	001000100	68
colonne 1	001000011	⊕ 100100100	101100111	359
colonne 2	001000011	⊕ 010010010	011010001	209
colonne 3	001000011	⊕ 001001001	000001010	10
diagonale 1	001000011	⊕ 100010001	101010010	338
diagonale 2	001000011	⊕ 001010100	000010111	23

Figure 3. Les coups possibles depuis la configuration `67` appliqués par un `xor`.

On suppose à disposition le dictionnaire `coups_possibles` qui énumère chaque coup représenté de la sorte :

```
coups_possibles = {
    "ligne 1"     : [1, 1, 1, 0, 0, 0, 0, 0, 0],
    "ligne 2"     : [0, 0, 0, 1, 1, 1, 0, 0, 0],
    "ligne 3"     : [0, 0, 0, 0, 0, 0, 1, 1, 1],
    "colonne 1"   : [1, 0, 0, 1, 0, 0, 1, 0, 0],
    "colonne 2"   : [0, 1, 0, 0, 1, 0, 0, 1, 0],
    "colonne 3"   : [0, 0, 1, 0, 0, 1, 0, 0, 1],
    "diagonale 1" : [1, 0, 0, 0, 1, 0, 0, 0, 1],
    "diagonale 2" : [0, 0, 1, 0, 1, 0, 1, 0, 0]
}
```

On souhaite implémenter une fonction qui prend en paramètre un entier `config` représentant une configuration et qui renvoie la liste des entiers correspondant à toutes les configurations accessibles en un seul coup.

```
def configurations_suivantes(config):
    config_bin = ...
    voisins = []
    for ... in ...:
        ... # ligne facultative
        ... = xor_etendu(..., ...)
        voisins.append(representant(...))
    return voisins
```

6. Recopier et compléter la fonction `configurations_suivantes`.

Partie B : graphe des configurations On constate qu'il semble impossible d'atteindre la victoire depuis la configuration représentée par l'entier `4`. On appelle configuration perdante toute configuration pour laquelle il n'existe aucune suite de coups menant à la victoire (c'est-à-dire à tous les jetons côté Recto). Inversement, on appelle configuration gagnante toute configuration qui n'est pas perdante.
La configuration représentée par l'entier `4` est effectivement perdante. Pour s'en convaincre on construit le graphe des configurations du jeu Recto-Verso.
Les sommets de ce graphe sont les différentes configurations possibles et il existe une arête $s_1 \to s_2$ d'un sommet s_1 vers un sommet s_2 si et seulement s'il est possible de passer de la configuration s_1 à la configuration s_2 en jouant un coup autorisé.
7. Montrer que s'il existe une arête $x \to y$ dans le graphe, alors il existe également une arête $y \to x$.
En Python, on choisit de représenter ce graphe par un dictionnaire construit grâce à la fonction `configurations_suivantes` :

```
graphe = {
  0 : [...],
  ...
  67 : [387, 123, 68, 359, 209, 10, 338, 23],
  ...
}
```

8. Une représentation par matrice d'adjacence aurait aussi pu être envisagée. Justifier le choix du dictionnaire en comparant pour les deux structures le nombre d'entiers nécessaires pour stocker en mémoire le graphe des configurations. On pourra laisser ce nombre écrit sous la forme d'une puissance de 2.

La fonction `parcours_profondeur` ci-dessous implémente un parcours en profondeur d'abord. Elle prend en paramètres la représentation d'un graphe, un entier `configuration` qui indique le sommet que l'on est en train de visiter, et une liste `vus` de sommets déjà visités par le parcours. Cette fonction effectue le parcours en rajoutant les sommets visités à la liste `vus`.

```
def parcours_profondeur(graphe, configuration, vus):
    vus.append(...)
    for s in ...:
        if s not in vus:
            parcours_profondeur(..., ..., ...)
```

9. Recopier et compléter la fonction `parcours_profondeur`.
10. En utilisant la fonction `parcours_profondeur`, écrire en Python une suite d'instructions permettant de vérifier que la configuration représentée par l'entier `4` est bien perdante.
11. Indiquer et justifier quel parcours de graphe (en largeur ou en profondeur) est le plus approprié pour trouver un chemin partant d'une configuration gagnante et menant à la victoire en un minimum de coups.

Corrigé

Partie A : représentation binaire des configurations **1.** Le plateau compte $3 \times 3 = 9$ jetons et chaque jeton a exactement deux états possibles (Recto ou Verso), indépendamment des huit autres. Une configuration est le choix d'un état pour chacun des neuf jetons : par le principe multiplicatif, il y a $2 \times 2 \times \cdots \times 2 = 2^9 = 512$ configurations. C'est aussi le nombre de mots de 9 bits, ce qui annonce la représentation binaire de la suite.
2. On note un plateau ligne par ligne, Recto $= 1$ et Verso $= 0$. La configuration de la figure 1 est `[0, 0, 1, 0, 0, 1, 1, 1, 0]`. On joue les trois coups, chacun retournant les trois jetons de la ligne ou de la colonne choisie :

Coup joué	**ligne 1**	**ligne 2**	**ligne 3**
(départ, figure 1)	0 0 1	0 0 1	1 1 0
après « ligne 1 »	1 1 0	0 0 1	1 1 0
après « ligne 2 »	1 1 0	1 1 0	1 1 0
après « colonne 2 »	1 0 0	1 0 0	1 0 0

La configuration finale `[1, 0, 0, 1, 0, 0, 1, 0, 0]` n'est pas la configuration victorieuse (les neuf jetons côté Recto) : la séquence ne mène pas à la victoire.
Ce que le correcteur attend : l'erreur porte sur le dernier coup. Après « ligne 1 » puis « ligne 2 », seule la colonne 3 est entièrement Verso ; le coup gagnant est « colonne 3 » et non « colonne 2 », qui retourne trois jetons déjà Recto. La séquence ligne 1, ligne 2, colonne 3 mène bien à la victoire (nous l'avons vérifié).
3. La liste de la figure 1, `[0, 0, 1, 0, 0, 1, 1, 1, 0]`, lue comme l'écriture binaire d'un entier (le premier bit est celui de poids fort, 2^8), donne :

$$0 \cdot 2^8 + 0 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 64 + 8 + 4 + 2 = 78.$$

La configuration de la figure 1 est représentée par l'entier 78. (Contrôle de la méthode sur l'exemple du sujet : `[0, 0, 1, 0, 0, 0, 0, 1, 1]` donne $64 + 2 + 1 = 67$.)
4. On parcourt la liste de gauche à droite : à chaque bit, l'entier déjà construit est décalé d'un rang (multiplié par 2) et le bit courant s'y ajoute (schéma de Horner). Cela évite de calculer des puissances de 2 et fonctionne quelle que soit la longueur de la liste.

```
def representant(liste):
```

```
    n = 0
    for bit in liste:
        n = 2 * n + bit
    return n
```

Trace sur `[0, 0, 1, 0, 0, 0, 0, 1, 1]` : `n` prend successivement les valeurs $0, 0, 1, 2, 4, 8, 16, 33, 67$; la fonction renvoie 67. Sur la liste de la figure 1 elle renvoie 78, sur `[1]*9` elle renvoie 511 et sur la liste vide elle renvoie 0.
Variante équivalente avec les puissances de 2, l'indice i portant le poids $2^{\texttt{len(liste)}-1-i}$:

```
def representant(liste):
    n = 0
    for i in range(len(liste)):
        n = n + liste[i] * 2 ** (len(liste) - 1 - i)
    return n
```

5. On applique `xor` indice par indice et l'on accumule les résultats dans une nouvelle liste, sans modifier les listes reçues :

```
def xor_etendu(l_x, l_y):
    resultat = []
    for i in range(len(l_x)):
        resultat.append(xor(l_x[i], l_y[i]))
    return resultat
```

Tests : `xor_etendu([1, 1], [0, 1])` renvoie `[1, 0]` (l'exemple du sujet) ; `xor_etendu([0, 0, 1, 0, 0, 0, 0, 1, 1], [1, 1, 1, 0, 0, 0, 0, 0, 0])` renvoie `[1, 1, 0, 0, 0, 0, 0, 1, 1]`, dont le représentant est 387 (figure 3) ; sur deux listes vides elle renvoie `[]`. La boucle fait `len(l_x)` tours : coût linéaire en la longueur des listes.
6. La configuration reçue est un entier : on la convertit d'abord en liste de neuf bits avec `binaire`. On parcourt ensuite les clés du dictionnaire (les noms des huit coups), on récupère pour chaque coup sa liste de bits (le « masque » des jetons à retourner), on l'applique par $\oplus$ et l'on convertit le résultat en entier avec `representant` :

```
def configurations_suivantes(config):
    config_bin = binaire(config)
    voisins = []
    for coup in coups_possibles:
        masque = coups_possibles[coup]        # ligne facultative
        resultat = xor_etendu(config_bin, masque)
        voisins.append(representant(resultat))
    return voisins
```

La ligne facultative peut être supprimée en écrivant directement `resultat = xor_etendu(config_bin, coups_possibles[coup])`. On peut aussi parcourir `coups_possibles.values()` et se passer de la clé. Comme un dictionnaire Python est parcouru dans l'ordre d'insertion de ses clés, l'appel `configurations_suivantes(67)` renvoie exactement `[387, 123, 68, 359, 209, 10, 338, 23]`, la liste de la figure 3 et de la ligne 67 du dictionnaire `graphe` (vérifié par exécution) ; `configurations_suivantes(4)` renvoie `[452, 60, 3, 288, 150, 77, 277, 80]`.

Partie B : graphe des configurations **7.** Soit une arête $x \to y$: il existe un coup, de masque c, tel que $y = x \oplus c$ (bit à bit). Jouons le même coup depuis y :

$$y \oplus c = (x \oplus c) \oplus c = x \oplus (c \oplus c) = x \oplus 0 = x,$$

car $\oplus$ est associatif, qu'un bit « ou exclusif » lui-même vaut 0 ($0 \oplus 0 = 0$ et $1 \oplus 1 = 0$) et que $\oplus 0$ laisse chaque bit inchangé. Concrètement, retourner deux fois de suite les mêmes jetons les remet dans leur état initial : chaque coup est sa propre opération inverse. Le coup c mène donc de y à x, et l'arête $y \to x$ existe. Le graphe est en fait non orienté (nous l'avons vérifié sur ses 512 sommets) ; chaque sommet a exactement 8 voisins.

8. Le graphe possède 2^9 sommets.

- Une matrice d'adjacence est un tableau carré à 2^9 lignes et 2^9 colonnes, soit $2^9 \times 2^9 = 2^{18} = 262\,144$ entiers (des 0 et des 1), quel que soit le nombre d'arêtes.
- Le dictionnaire associe à chacun des 2^9 sommets la liste de ses 8 voisins, donc $2^9 \times 8 = 2^9 \times 2^3 = 2^{12} = 4\,096$ entiers dans les listes, auxquels s'ajoutent les $2^9 = 512$ clés : $2^{12} + 2^9 = 4\,608$ entiers en tout.

La matrice occupe environ 64 fois plus de mémoire ($2^{18}/2^{12} = 2^6$), et elle est presque entièrement remplie de 0 : chaque sommet n'a que 8 voisins sur 512 possibles, le graphe est creux. Le dictionnaire (listes d'adjacence) ne stocke que les arêtes qui existent ; de plus, parcourir les voisins d'un sommet y coûte 8 opérations au lieu de 512 (une ligne entière de la matrice). C'est le choix adapté.

9. Le sommet en cours de visite est ajouté à `vus`, puis on visite récursivement chacun de ses voisins non encore vus ; les voisins sont lus dans le dictionnaire :

```
def parcours_profondeur(graphe, configuration, vus):
    vus.append(configuration)
    for s in graphe[configuration]:
        if s not in vus:
            parcours_profondeur(graphe, s, vus)
```

La récursion termine car chaque appel ajoute un nouveau sommet à `vus` et que le graphe est fini (au plus 512 appels, profondeur de pile bien inférieure à la limite de Python) ; le test `s not in vus` empêche de tourner en rond dans les cycles.

10. La configuration victorieuse a ses neuf jetons côté Recto : c'est la liste `[1]*9`, soit l'entier $2^9 - 1 = 511$. Une configuration est perdante si aucune suite de coups ne mène à la victoire, c'est-à-dire si 511 n'est pas atteignable depuis elle dans le graphe : on lance le parcours depuis 4 et l'on regarde si 511 figure parmi les sommets vus.

```
victoire = representant([1, 1, 1, 1, 1, 1, 1, 1, 1])   # 511
vus = []
parcours_profondeur(graphe, 4, vus)
print(victoire not in vus)   # affiche True : 4 est perdante
```

On peut aussi écrire `assert 511 not in vus`. Exécuté sur le graphe construit avec `configurations_suivantes`, le parcours depuis 4 atteint 128 configurations, et 511 n'en fait pas partie : la configuration 4 est bien perdante. (Depuis 67, au contraire, les 128 configurations atteintes contiennent 511.)

Ce que le correcteur attend : la liste `vus` doit être créée vide *avant* l'appel et passée en argument, puisque c'est la fonction qui la remplit ; l'argument est l'entier 4, pas la liste `binaire(4)`.

11. Le parcours en largeur est le plus approprié. Il explore le graphe par « couches » à l'aide d'une file : d'abord la configuration de départ, puis toutes celles accessibles en un coup, puis celles accessibles en deux coups, etc. Comme toutes les arêtes ont le même poids (un coup), la première fois que la victoire est rencontrée, c'est nécessairement à la distance minimale : le chemin reconstitué a le minimum de coups.

Le parcours en profondeur, lui, s'enfonce le plus loin possible le long d'une branche avant de revenir en arrière : il atteint la victoire par le premier chemin qu'il trouve, en général très

long. Par exemple, depuis la configuration 78 de la figure 1, le parcours en profondeur écrit à la question 9 atteint 511 après 23 coups, alors que le parcours en largeur donne le chemin minimal en 3 coups : ligne 1, ligne 2, colonne 3 (question 2).

Exercice 3 (8 points)

Sujet — Exercice 3 — 8 points

Cet exercice porte sur les arbres, l'algorithmique des arbres, et les bases de données.
On s'intéresse au fonctionnement interne d'une plateforme en ligne dédiée spécifiquement aux débats. Plutôt que de lister des messages des utilisateurs dans l'ordre chronologique, les différentes contributions sont structurées de façon arborescente afin de mieux comprendre les liens entre les arguments. Chaque contribution vient se rattacher à une affirmation, soit afin d'appuyer cette affirmation avec un argument pour, soit afin de l'attaquer avec un argument contre.
Cet exercice contient deux parties indépendantes l'une de l'autre.

Partie A : structure arborescente des arguments Dans cette partie on modélise un arbre de débat pour un sujet de débat. La figure 1 montre un exemple d'arbre de débat.

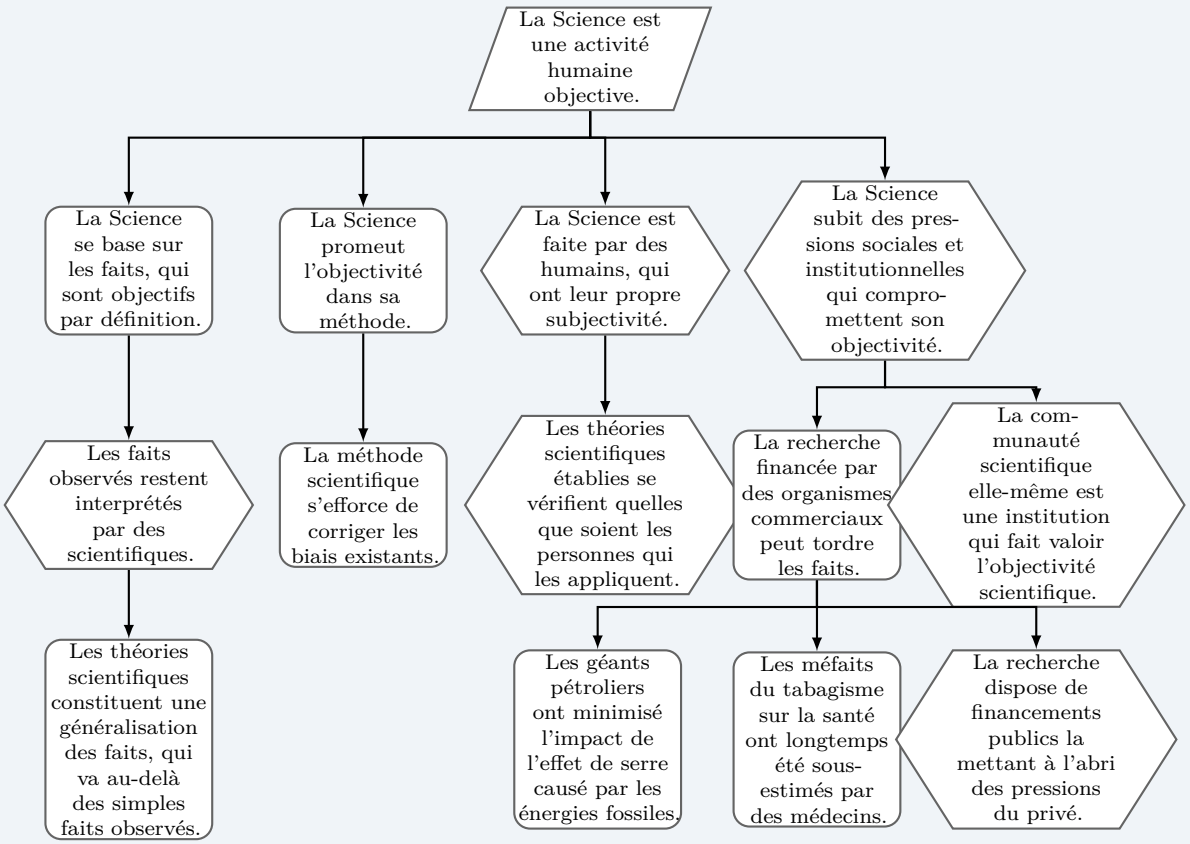

Figure 1. Arbre de débat sur l'objectivité de la science

Dans un arbre de débat, chaque nœud est une affirmation, et il y en a de trois sortes :

— le sujet du débat (dans le parallélogramme) est une affirmation, c'est le point de départ du débat ;

— un argument pour (dans un rectangle aux coins arrondis) peut venir soutenir une affirmation, cet argument est lui-même une affirmation ;

— un argument contre (dans un hexagone) peut venir attaquer une affirmation, cet argument est lui-même une affirmation.

De plus, chaque affirmation peut être likée (j'aime) ou dislikée (je n'aime pas) par les utilisateurs.
1. Indiquer le nom donné au nœud « point de départ du débat » dans le vocabulaire des arbres.
2. Indiquer en justifiant si un arbre de débat est un arbre binaire ou non.
On modélise la structure d'arbre de débat à l'aide de la classe suivante :

```
1  class Affirmation:
2      def __init__(self, phrase):
3          """
4          Création d'un objet Affirmation à partir d'une
5          phrase (str).
6          """
7          self.contenu = phrase
8          self.sorte = "" # vaudra "sujet", "pour" ou "contre"
9                          # vaut "" avant ajout dans l'arbre
10         self.arguments = []
11         self.nb_likes = 0
12         self.nb_dislikes = 0
```

3. Donner le nom et le type de chaque attribut de la classe `Affirmation`.
On souhaite ajouter à cette classe une méthode `soutenir` qui permet d'ajouter un argument pour à l'affirmation.
On suppose à disposition une méthode `contrer` similaire qui permet d'ajouter un argument contre à l'affirmation.

```
1  def soutenir(self, argument):
2      """
3      Ajoute l'Affirmation `argument` comme argument pour de
4      l'Affirmation `self`.
5
6      Précondition : l'Affirmation ne doit pas avoir déjà été
7      utilisée, ni comme sujet, ni comme pour, ni comme
8      contre, ce qu'on vérifie grâce à son attribut `sorte`.
9      """
10     assert argument.sorte == ...
11     ... .append(argument)
12     argument.sorte = "..."
```

4. Recopier et compléter les lignes 10 à 12 de la méthode `soutenir`, en respectant sa documentation.
Pour mesurer les débats les plus vifs, la plateforme de débats dispose de plusieurs mesures.
5. Écrire la méthode `nb_contre` qui permet d'obtenir le nombre d'arguments contre directement rattachés à cette `Affirmation`.
On s'intéresse à la méthode `mystere` suivante, qui utilise la méthode `nb_contre` :

```
1  def mystere(self):
2      m = self.nb_contre()
3      for arg in self.arguments:
```

```
        candidat = arg.mystere()
        if candidat > m:
            m = candidat
    return m
```

6. Expliquer par une phrase ce que renvoie la méthode `mystere`.
Afin de trancher le débat dans sa globalité, on souhaite évaluer une `Affirmation`, en tenant compte de la totalité des likes et des dislikes de chaque argument, et en tenant compte du fait que ces arguments soient pour ou contre. Si l'évaluation est positive le pour l'emporte, si elle est négative le contre l'emporte.
La formule pour l'évaluation d'une affirmation correspond à la somme de ses likes et des évaluations de ses arguments pour, à laquelle on soustrait ses dislikes et les évaluations de ses arguments contre.
7. Écrire, en utilisant uniquement des lignes de code choisies parmi les lignes suivantes, une méthode `evaluation` qui renvoie le résultat du calcul décrit dans la formule ci-dessus. Les lignes ne sont pas toutes à utiliser et l'indentation est à adapter.

```
def evaluation(self):
def evaluation():
total = self.nb_likes + self.nb_dislikes
total = self.nb_likes - self.nb_dislikes
total = self.nb_dislikes - self.nb_likes
total = self.nb_likes * self.nb_dislikes
for arg in self.arguments:
for arg in self.arguments_pour:
for arg in self.arguments_contre:
if arg.sorte == "sujet":
if arg.sorte == "pour":
if arg.sorte == "contre":
else:
total = arg.evaluation()
total = total + arg.evaluation()
total = total - arg.evaluation()
total = total * arg.evaluation()
return total
return 1 + total
```

Partie B : base de données des utilisateurs et de leurs contributions Dans cette partie, 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`, `OR`) et `JOIN ... ON` ;
- construire des requêtes d'insertion et de mise à jour à l'aide de `UPDATE`, `INSERT` et `DELETE`.

Pour manipuler toutes les informations sur plusieurs sujets de débats, la plateforme dispose d'un Système de Gestion de Bases de Données (SGBD).
8. Parmi les propositions suivantes, indiquer celles qui font partie des rôles d'un SGBD :

a) sécuriser les accès à la base de données ;
b) assurer l'alimentation électrique des serveurs ;
c) assurer la persistance des données même en cas de panne matérielle ;
d) gérer les accès en parallèle de plusieurs utilisateurs ;
e) assurer des connexions en HTTPS au serveur.

Pour gérer les données, le schéma relationnel utilisé est représenté sur la figure 2 (transcrit ici en notation textuelle : clés primaires soulignées, clés étrangères précédées d'un « # ») :

`utilisateur`(<u>`pseudo`</u>, `email`, `nom`, `prenom`)
`affirmation`(<u>`id_aff`</u>, `contenu`, `#auteur`, `sorte`, `#aff_repondue`, `nb_likes`, `nb_dislikes`)
`reaction`(<u>`#aff_reagie`</u>, <u>`#utilisateur`</u>, `sorte`)

Flèches du schéma : `affirmation.auteur` et `reaction.utilisateur` référencent `utilisateur.pseudo` ; `affirmation.aff_repondue` et `reaction.aff_reagie` référencent `affirmation.id_aff`.

Figure 2. Schéma relationnel de la plateforme de débat

La clé primaire de chaque table correspond à l'ensemble de ses attributs soulignés. Chaque attribut précédé d'un « # » est une clé étrangère.
L'attribut `sorte` de la table `affirmation` peut valoir `'sujet'` quand c'est l'affirmation initiale d'un sujet de débat, ou bien `'pour'` ou `'contre'` quand c'est une réponse à une autre affirmation (référencée par l'attribut `aff_repondue`). L'attribut `contenu` correspond à la phrase elle-même de l'affirmation.
L'attribut `sorte` de la table `reaction` peut valoir `'like'` ou `'dislike'`, et s'applique à l'affirmation référencée par l'attribut `aff_reagie`.
Voici des exemples d'extraits des tables `affirmation` et `reaction`. On a omis, pour ces exemples, les attributs `auteur` et `contenu` de la table `affirmation` :

`affirmation`

id_aff	sorte	aff_repondue	nb_likes	nb_dislikes
0	'sujet'	NULL	42	17
1	'pour'	0	20	20
2	'pour'	0	40	10
3	'contre'	0	13	12
4	'contre'	0	18	6
7	'contre'	1	15	16

`reaction`

aff_reagie	utilisateur	sorte
0	'alice'	'like'
1	'alice'	'dislike'
0	'bob'	'dislike'
1	'bob'	'dislike'

9. Indiquer en justifiant quel autre attribut de la table `utilisateur` aurait aussi pu servir de clé primaire.
On rappelle que la fonction d'agrégation `COUNT` permet de compter les éléments du résultat d'une requête en plaçant `COUNT(*)` dans la clause `SELECT`.
10. Écrire une requête SQL permettant d'obtenir le nombre d'affirmations qui ont 50 likes ou plus.
11. Écrire une requête qui permet d'obtenir les contenus des sujets et leur nombre de likes, créés par les utilisateurs dont les prénom et nom sont respectivement `'Pierre'` et `'Durand'`.
Le mot-clé `AS` permet de donner un autre nom à une table au sein d'une requête. Il est notamment nécessaire lorsqu'on a besoin de faire la jointure d'une table avec elle-même. On considère la requête SQL suivante :

```sql
SELECT aff.contenu, rep.contenu
FROM affirmation AS aff JOIN affirmation AS rep
  ON rep.affirmation_repondue = aff.id_affirmation
WHERE rep.nb_likes >= 2 * aff.nb_likes
  AND rep.sorte = 'contre'
```

12. Donner une interprétation en français de ce que permet d'obtenir la requête ci-dessus.
L'utilisateur au pseudo `'i<3descartes'` approuve par un like l'affirmation `'Cogito ergo sum'` qui a l'`id_aff` 108.
13. Recopier et compléter les deux requêtes SQL suivantes, permettant de mettre à jour les informations de la base de données suite à cette action.

```sql
... reaction
VALUES (..., ..., ...);;

... affirmation
SET nb_likes = ...
WHERE ... = ...;;
```

L'utilisateur au pseudo `'i<3rgpd'` souhaite faire valoir son *droit à l'effacement*, et supprimer son compte utilisateur et toutes les données qui y sont liées. Dans cette situation, la plateforme de débat, qui souhaite maintenir la qualité des échanges, suit la procédure suivante :

— elle anonymise toutes les affirmations de l'utilisateur (en remplaçant l'attribut `auteur` par la valeur `NULL`), considérant que les affirmations elle-mêmes ne sont pas des données personnelles ;

— elle supprime toutes les réactions de l'utilisateur (mais sans toucher au nombres de likes et de dislikes des affirmations correspondantes) ;

— et enfin elle supprime l'utilisateur lui-même.

14. Expliquer pourquoi il est nécessaire de supprimer les réactions de l'utilisateur avant de supprimer l'utilisateur.
15. Écrire les trois requêtes qui permettent la suppression des données du compte de `'i<3rgpd'` en suivant la procédure décrite ci-dessus.

Corrigé

Partie A : structure arborescente des arguments **1.** Le point de départ du débat est la *racine* de l'arbre : l'unique nœud sans parent, dont tous les autres nœuds descendent. Sur la figure 1, c'est le parallélogramme « La Science est une activité humaine objective. » ; les arguments sans réponse (par exemple « Les géants pétroliers… ») en sont les feuilles.
2. Un arbre binaire est un arbre dont chaque nœud possède *au plus deux* enfants (un sous-arbre gauche et un sous-arbre droit). Ce n'est pas le cas d'un arbre de débat : une affirmation peut recevoir autant d'arguments qu'on veut. Sur la figure 1, la racine a quatre enfants (deux arguments pour et deux arguments contre) et l'affirmation « La recherche financée par des organismes commerciaux peut tordre les faits. » en a trois. Un arbre de débat est donc un arbre général (dit « quelconque »), et non un arbre binaire. Dans la classe `Affirmation`, cela se traduit par l'attribut `arguments`, une liste d'enfants de longueur quelconque, là où un arbre binaire aurait deux attributs `gauche` et `droit`.
3. Les attributs sont ceux créés dans le constructeur `__init__` :

Attribut	Type	Rôle
`contenu`	`str`	la phrase de l'affirmation
`sorte`	`str`	`""`, puis `"sujet"`, `"pour"` ou `"contre"`
`arguments`	`list` (d'objets `Affirmation`)	les arguments rattachés, vide au départ
`nb_likes`	`int`	nombre de likes, 0 au départ
`nb_dislikes`	`int`	nombre de dislikes, 0 au départ

4. La documentation impose de vérifier que l'argument n'a encore jamais été utilisé : avant tout ajout, son attribut `sorte` vaut la chaîne vide (ligne 8 du constructeur). On l'ajoute ensuite à la liste des arguments de `self`, puis on le marque comme argument pour :

```
10      assert argument.sorte == ""
11      self.arguments.append(argument)
12      argument.sorte = "pour"
```

Testé : après `s.soutenir(a)`, `a` figure dans `s.arguments` et `a.sorte` vaut `"pour"` ; un second `s.soutenir(a)` lève une `AssertionError`, conformément à la précondition.
Ce que le correcteur attend : c'est `self.arguments` qui reçoit l'argument (et non `argument.arguments`), et c'est `argument.sorte` qui change (pas `self.sorte`, qui reste ce qu'il était).
5. On parcourt les arguments directement rattachés (la liste `arguments`, sans descendre plus bas dans l'arbre) et l'on compte ceux dont la sorte est `"contre"` :

```
def nb_contre(self):
    n = 0
    for arg in self.arguments:
        if arg.sorte == "contre":
            n = n + 1
    return n
```

Sur l'arbre de la figure 1 (reconstruit et exécuté) : la racine renvoie 2, « La Science se base sur les faits… » renvoie 1, et une feuille renvoie 0.
6. La méthode `mystere` renvoie le plus grand nombre d'arguments contre directement rattachés à une même affirmation, parmi l'affirmation `self` et toutes celles de son sous-arbre (ses arguments, les arguments de ses arguments, etc.) : c'est le « maximum de contre-attaques directes » subies par une affirmation du débat.
En effet, `m` part de `self.nb_contre()`, puis, pour chaque argument, l'appel récursif `arg.mystere()` fournit le maximum sur le sous-arbre de cet argument, et `m` ne conserve que le plus grand de tous ces candidats. La récursion s'arrête d'elle-même sur les feuilles, dont la boucle ne fait aucun tour. Sur la figure 1, `mystere` appelée sur la racine renvoie 2 (la racine subit deux arguments contre, aucune autre affirmation n'en subit plus d'un).
7. L'évaluation vaut (likes − dislikes) de l'affirmation, plus l'évaluation de chaque argument pour, moins l'évaluation de chaque argument contre ; c'est une définition récursive, l'appel `arg.evaluation()` descendant dans le sous-arbre :

```
def evaluation(self):
    total = self.nb_likes - self.nb_dislikes
    for arg in self.arguments:
        if arg.sorte == "pour":
            total = total + arg.evaluation()
        else:
            total = total - arg.evaluation()
    return total
```

Le `else` est légitime car un argument rattaché ne peut être que `"pour"` ou `"contre"` ; on peut aussi remplacer `else:` par `if arg.sorte == "contre":`. Les autres lignes proposées sont à écarter : `def evaluation():` n'a pas le paramètre `self` ; `nb_likes + nb_dislikes` et le produit ne suivent pas la formule ; `arguments_pour` et `arguments_contre` ne sont pas des attributs de la classe ; `if arg.sorte == "sujet"` ne peut jamais être vrai pour un argument ; `total = arg.evaluation()` écraserait le total au lieu de l'accumuler ; et `return 1 + total` ajouterait 1 à chaque nœud.
Vérification par exécution sur l'extrait de table de la partie B, qui décrit un petit arbre (0 est le sujet, 1 et 2 ses arguments pour, 3 et 4 ses arguments contre, 7 un argument contre de 1) : `evaluation`$(7) = 15 - 16 = -1$; `evaluation`$(1) = (20 - 20) - (-1) = 1$; `evaluation`$(2) = 30$; `evaluation`$(3) = 1$; `evaluation`$(4) = 12$; d'où `evaluation`$(0) = (42 - 17) + 1 + 30 - 1 - 12 = 43 > 0$: le pour l'emporte.

Partie B : base de données des utilisateurs et de leurs contributions **8.** Les rôles d'un SGBD sont **a)**, **c)** et **d)**.

- a) Oui : le SGBD gère les droits d'accès (qui peut lire, modifier, supprimer quelles tables) et authentifie les utilisateurs de la base.
- b) Non : l'alimentation électrique relève de l'infrastructure matérielle (onduleurs, salle serveurs), pas d'un logiciel.
- c) Oui : la persistance est le cœur du SGBD ; il écrit les données sur disque et garantit, grâce à ses journaux et ses transactions, qu'une panne en cours d'opération ne laisse pas la base dans un état incohérent.
- d) Oui : le SGBD gère les accès concurrents de plusieurs utilisateurs (ou programmes) sans corrompre les données.
- e) Non : HTTPS est un protocole de la couche application du web, assuré par le serveur web ; le SGBD n'y intervient pas.

9. L'attribut `email` aurait aussi pu servir de clé primaire : une adresse électronique est propre à une personne, deux comptes ne peuvent pas partager la même, et elle est nécessairement renseignée à l'inscription ; elle identifie donc chaque n-uplet de façon unique et sans valeur `NULL`, ce qu'exige une clé primaire. Ni `nom` ni `prenom` ne conviennent : plusieurs utilisateurs peuvent s'appeler Pierre Durand.
10. On compte les lignes de `affirmation` dont le nombre de likes est au moins 50 :

```sql
SELECT COUNT(*)
FROM affirmation
WHERE nb_likes >= 50;
```

Sur l'extrait donné, le résultat serait 0 (le maximum est 42) ; vérifiée sur une base d'essai, la requête renvoie bien le nombre de lignes à `nb_likes` ≥ 50.
11. Le nom et le prénom sont dans `utilisateur`, le contenu et les likes dans `affirmation` : il faut une jointure sur la clé étrangère `auteur`, qui référence `pseudo`, puis filtrer les sujets et l'identité de l'auteur :

```sql
SELECT affirmation.contenu, affirmation.nb_likes
FROM affirmation
JOIN utilisateur ON affirmation.auteur = utilisateur.pseudo
WHERE affirmation.sorte = 'sujet'
  AND utilisateur.prenom = 'Pierre'
  AND utilisateur.nom = 'Durand';
```

Ce que le correcteur attend : la condition `sorte = 'sujet'` (on veut les sujets, pas tous les messages de Pierre Durand), la jointure (le nom n'est pas dans `affirmation`) et les deux conditions liées par `AND`.

12. La table `affirmation` est jointe avec elle-même : `aff` désigne une affirmation et `rep` une réponse qui lui est rattachée (sa clé étrangère « affirmation répondue » vaut l'identifiant de `aff`). Les conditions gardent les réponses de sorte `'contre'` dont le nombre de likes est au moins le double de celui de l'affirmation attaquée. La requête renvoie donc, pour chaque argument contre au moins deux fois plus liké que l'affirmation qu'il attaque, le contenu de cette affirmation et le contenu de l'argument contre : elle repère les contre-arguments nettement plus populaires que ce qu'ils contredisent.

Remarque : la requête imprimée nomme les attributs `affirmation_repondue` et `id_affirmation`, alors que le schéma de la figure 2 les appelle `aff_repondue` et `id_aff` ; avec les noms du schéma, la jointure s'écrit `ON rep.aff_repondue = aff.id_aff`. Sur l'extrait de table, aucune ligne ne ressort : l'argument 7 (15 likes) attaque l'affirmation 1 (20 likes), et 3 et 4 (13 et 18 likes) attaquent 0 (42 likes) ; aucun n'atteint le double.

13. On ajoute une réaction (les valeurs dans l'ordre des attributs de la table : `aff_reagie`, `utilisateur`, `sorte`) puis on incrémente le compteur de likes de l'affirmation 108 :

```sql
INSERT INTO reaction
VALUES (108, 'i<3descartes', 'like');

UPDATE affirmation
SET nb_likes = nb_likes + 1
WHERE id_aff = 108;
```

Testé sur une base d'essai : la ligne `(108, 'i<3descartes', 'like')` apparaît dans `reaction` et `nb_likes` de l'affirmation 108 passe de 60 à 61.

Ce que le correcteur attend : `nb_likes = nb_likes + 1` (on ne connaît pas la valeur actuelle, on ne peut pas écrire un nombre), et la clause `WHERE` sur `id_aff`, sans laquelle toutes les affirmations gagneraient un like.

14. Dans la table `reaction`, l'attribut `utilisateur` est une clé étrangère qui référence `utilisateur.pseudo`. La contrainte d'intégrité référentielle impose que toute valeur d'une clé étrangère corresponde à une clé primaire existante. Si l'on supprimait d'abord l'utilisateur, ses réactions référenceraient un pseudo qui n'existe plus : le SGBD refuse cette suppression (erreur de contrainte de clé étrangère, ce que nous avons observé sur une base d'essai). Il faut donc d'abord faire disparaître toutes les références au pseudo (les réactions, et l'attribut `auteur` des affirmations, mis à `NULL`), puis seulement supprimer l'utilisateur.

15. Les trois requêtes, dans l'ordre de la procédure :

```sql
UPDATE affirmation
SET auteur = NULL
WHERE auteur = 'i<3rgpd';

DELETE FROM reaction
WHERE utilisateur = 'i<3rgpd';

DELETE FROM utilisateur
WHERE pseudo = 'i<3rgpd';
```

La première anonymise les affirmations sans les supprimer (leurs `nb_likes` et `nb_dislikes` sont intacts) ; la deuxième supprime les réactions sans toucher aux compteurs des affirmations, comme demandé ; la troisième supprime enfin le compte, ce que le SGBD accepte maintenant

que plus rien ne le référence. Exécutées dans cet ordre sur une base d'essai, elles laissent les affirmations de `'i<3rgpd'` avec un auteur `NULL` et leurs compteurs inchangés, et suppriment l'utilisateur.