Numérique et sciences informatiques

Baccalauréat général 2025 — épreuve de spécialité

Métropole, session 2025 (25-NSIJ2ME1)

Corrigé

Trois exercices

Ibrahim Alame — Adloun, édition 2026

Exercice 1 (6 points)

Sujet — Exercice 1 — 6 points

Cet exercice porte sur les arbres binaires et la programmation Python.
Le codage de Shannon-Fano est un système de codage utilisé pour la compression sans pertes de données. Il a été mis au point par Robert Fano d'après une idée de Claude Shannon.

Partie A Dans cette partie, on va étudier l'utilisation des arbres de codage.
Un arbre de codage est un arbre binaire où chaque feuille contient un symbole du texte que l'on souhaite coder. Le code binaire d'un symbole s'obtient alors en concaténant les `0` et les `1` sur les branches qui mènent de la racine à la feuille contenant ce symbole. Par exemple, pour l'arbre de codage donné en Figure 1, le symbole `c` est codé par le mot binaire `1101`, tandis que `d` est codé par le mot binaire `11000`. Les codes binaires des symboles ne sont donc pas tous de la même taille. Pour décoder un mot binaire, il suffit de descendre dans l'arbre, depuis la racine, selon les `0` et les `1` qu'on lit jusqu'à trouver une feuille (et donc un symbole), puis de recommencer avec la suite du mot binaire pour décoder les symboles suivants.

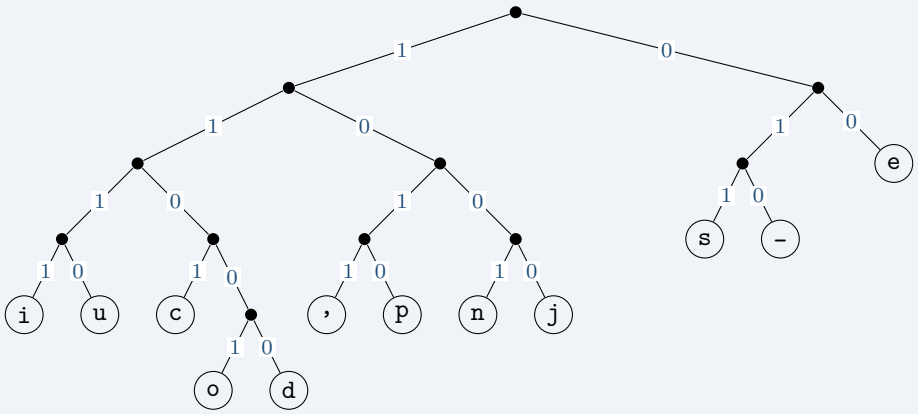


Figure 1. Exemple d'arbre de codage

1. Écrire le mot binaire qui sera utilisé pour encoder le caractère espace, représenté par le symbole `_` dans l'arbre.
2. Déterminer le texte codé par le mot binaire `000111010111111100011001`.
3. Citer le type de parcours de l'arbre qui permettrait d'obtenir les symboles classés par taille d'encodage croissante.

Partie B Dans cette partie, on va utiliser le codage de Shannon-Fano pour encoder le texte : `je pense, donc je suis`
Dans la méthode de Shannon-Fano, l'arbre de codage est calculé pour un texte donné par l'algorithme suivant.

— Étape 1 : classer les symboles du texte par nombre d'occurrences croissant ;

— Étape 2 : en gardant le classement obtenu, séparer les symboles en deux sous-groupes de sorte que les totaux des nombres d'occurrences soient les plus proches possibles dans les deux sous-groupes ;

— Étape 3 : placer tous les symboles du premier groupe dans le fils gauche (côté étiqueté par `1`), et ceux du second groupe dans le fils droit (côté étiqueté par `0`) ;

— Étape 4 : recommencer récursivement pour chacun des sous-groupes jusqu'à ce qu'ils n'aient plus qu'un seul symbole ; on a alors une feuille étiquetée par ce symbole.

Après avoir classé les symboles par nombre d'occurrences croissant (étape 1), on obtient le tableau suivant :

symbole	`i`	`u`	`c`	`o`	`d`	`,`	`p`	`n`	`j`	`s`	`_`	`e`
nombre d'occurrences	1	1	1	1	1	1	1	2	2	3	4	4

4. Justifier par le calcul que l'étape 2 mène à la situation illustrée par la Figure 2.

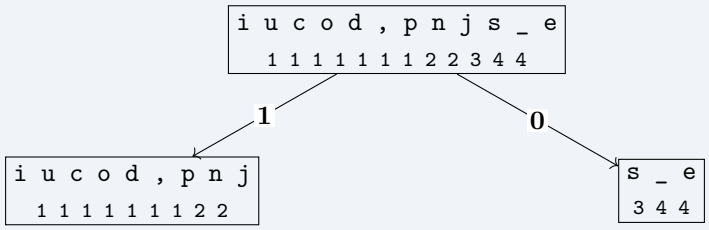


Figure 2. Le résultat de l'étape 2

En appliquant l'algorithme de Shannon-Fano, on peut obtenir l'arbre de la Figure 3.

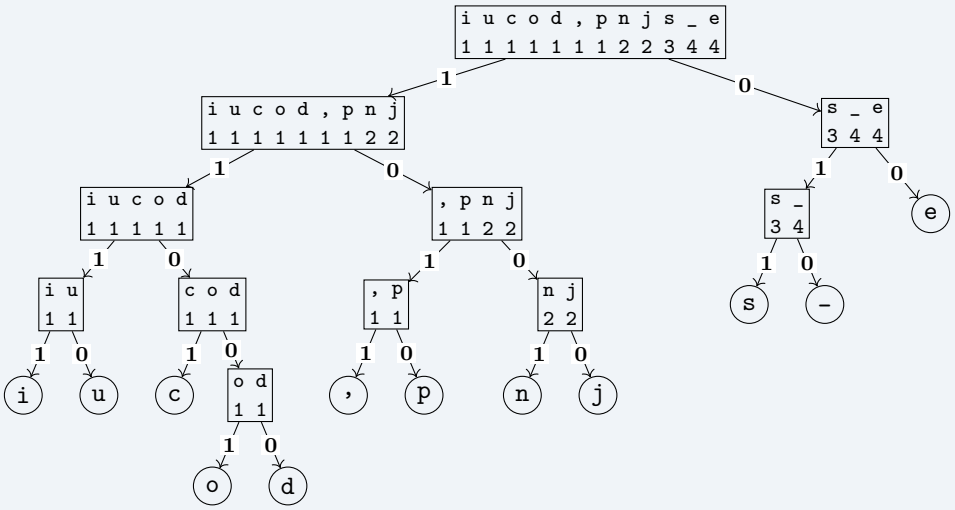


Figure 3. Arbre de codage obtenu par l'algorithme de Shannon-Fano

On rappelle qu'un arbre réduit à un seul nœud, c'est-à-dire réduit à une feuille, est de hauteur 0.

5. Donner la hauteur de l'arbre de la Figure 3 et préciser dans le contexte de l'exercice ce qu'elle représente.

On rappelle que dans le code ASCII, chaque symbole est codé sur un octet.

6. Justifier, en comparant le codage ASCII et le codage de Shannon-Fano, que ce second codage permet d'utiliser environ deux fois moins d'octets pour le texte :
`je pense, donc je suis`

7. Dessiner, en vous inspirant de l'arbre de la Figure 1, un arbre de codage qui permettrait d'encoder le mot « `chiffrer` » en utilisant l'algorithme de Shannon-Fano.

Partie C Dans cette partie, on souhaite écrire une fonction Python qui donnera le mot binaire obtenu pour coder un texte avec l'algorithme de Shannon-Fano. On commence par la fonction `creer_dico_occ` :

```
1  def creer_dico_occ(texte):
2      """renvoie un dictionnaire dont les clés sont les
3      symboles de texte et les valeurs associées leur
4      nombre d'occurences dans texte"""
5      dico = {}
6      for symbole in texte:
7          if symbole in dico:
8              dico[symbole] = ...
9          else:
10             dico[symbole] = ...
11     return dico
```

8. Recopier et compléter les lignes 8 et 10 du code de la fonction `creer_dico_occ`.
On dispose d'une fonction `creer_tab_trie` qui prend en paramètre un dictionnaire construit avec la fonction `creer_dico_occ` et qui renvoie une liste de tuples classés dans l'ordre croissant d'occurrences des symboles.
Par exemple :

```
>>> texte = 'je pense, donc je suis'
>>> dico = creer_dico_occ(texte)
>>> creer_tab_trie(dico)
[('i', 1), ('u', 1), ('c', 1), ('o', 1), ('d', 1), (',', 1),
('p', 1), ('n', 2), ('j', 2), ('s', 3), (' ', 4), ('e', 4)]
```

9. Écrire une fonction `somme_occ` qui prend en paramètres un tableau `tab` de tuples `(symbole, nb_occ)` et qui renvoie la somme des nombres d'occurrences des symboles du tableau. Les tuples utilisés sont de même structure que l'élément renvoyé dans l'exemple précédent.
On suppose pour la suite qu'on dispose d'une fonction `separe` qui sépare un tableau trié en deux sous-tableaux de manière à ce que les sommes de ces derniers soient les plus proches possible :

```
1  def separe(tab):
2      moitie = somme_occ(tab) // 2
3      somme = 0
4      i = 0
5      while moitie > somme:
6          somme = somme + tab[i][1]
7          i = i + 1
8      tab1 = [tab[k] for k in range(0, i)]
9      tab2 = [tab[k] for k in range(i, len(tab))]
10     return tab1, tab2
```

10. Recopier et compléter les lignes 9 et 11 du code de la fonction récursive `shannon` qui prend en paramètres un caractère `symbole` et un tableau trié `tab` et qui renvoie l'écriture binaire associée à `symbole` dans le tableau `tab`.

```
1  def shannon(symbole, tab):
2      """renvoie l'écriture binaire associée à symbole
3      dans le tableau trié tab"""
4      if len(tab) == 1:
```

```
        return ""
    else:
        t1, t2 = separe(tab)
        if symbole in [elt[0] for elt in t1]:
            return "1" + ...
        else:
            return "0" + ...
```

11. Décrire ce qui garantit la terminaison de la fonction récursive `shannon`.
12. Écrire une fonction `encode_shannon` qui prend en paramètre un texte de type `str` et renvoie un mot binaire de type `str` obtenu après encodage par l'algorithme de Shannon-Fano. On pourra utiliser les fonctions vues précédemment qui sont recensées ci-après.

```
creer_dico_occ(texte)
  renvoie un dictionnaire dont les clés sont les symboles
  du texte et les valeurs associées leur nombre
  d'occurrences

creer_tab_trie(dico)
  renvoie la liste crée à partir d'un dictionnaire
  de couples (symbole, nb_occ)

separe(tab)
  renvoie le tuple composé des 2 sous-tableaux triés
  avec des sommes d'occurences proches

shannon(symbole, tab)
  renvoie l'écriture binaire associée au symbole dans le
  tableau trié tab
```

Corrigé

Partie A **1.** On lit les étiquettes des branches depuis la racine jusqu'à la feuille `_` : branche `0` (vers la droite), puis `1`, puis `0`. Le caractère espace est donc codé par le mot binaire `010`.
On vérifie sur les exemples du sujet : `c` s'obtient par `1`, `1`, `0`, `1` soit `1101`, et `d` par `1`, `1`, `0`, `0`, `0` soit `11000` : c'est bien la lecture attendue.
2. On descend depuis la racine en suivant les bits, et l'on repart de la racine chaque fois qu'une feuille est atteinte :

bits lus	chemin	symbole
00	droite, droite	e
011	droite, gauche, gauche	s
1010	gauche, droite, gauche, droite	p
1111	gauche, gauche, gauche, gauche	i
11001	gauche, gauche, droite, droite, gauche	o
1001	gauche, droite, droite, gauche	n

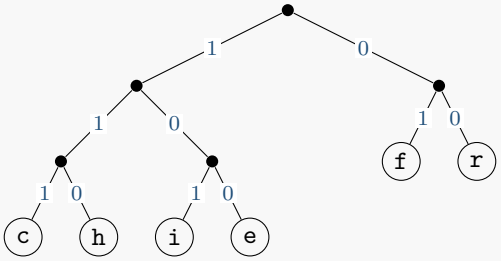
Le mot binaire 00 011 1010 1111 11001 1001 se décode donc en « `espion` ». Les 22 bits ont tous été consommés et la dernière lecture s'achève exactement sur une feuille : le décodage est complet.
Ce que le correcteur attend : le décodage n'est pas ambigu parce qu'aucun code n'est le préfixe d'un autre (les symboles ne sont que sur les feuilles) : dès qu'on atteint une feuille, on sait que le symbole est terminé et on repart de la racine.

3. La taille de l'encodage d'un symbole est la profondeur de sa feuille (le nombre de branches entre la racine et la feuille). Le parcours qui visite les nœuds niveau par niveau, par profondeur croissante, est le **parcours en largeur** (BFS) : il rencontre d'abord `e` (profondeur 2), puis `s` et `_` (profondeur 3), puis `i`, `u`, `c`, la virgule, `p`, `n`, `j` (profondeur 4), enfin `o` et `d` (profondeur 5). Il s'implémente avec une file (FIFO) : on défile un nœud, on enfile ses enfants. Un parcours en profondeur (préfixe, infixe ou suffixe) ne convient pas : il descend jusqu'aux feuilles d'un sous-arbre avant de passer au suivant et mélangerait les profondeurs.

Partie B **4.** Le total des occurrences est $7 \times 1 + 2 + 2 + 3 + 4 + 4 = 22$, ce qui correspond bien aux 22 caractères du texte (espaces et virgule compris). La moitié vaut 11. On cumule les occurrences dans l'ordre du tableau : $1, 2, 3, 4, 5, 6, 7$ après les sept symboles à une occurrence, puis 9 avec `n`, puis 11 avec `j`. Le premier groupe {`i`, `u`, `c`, `o`, `d`, « `,` », `p`, `n`, `j`} totalise 11 occurrences et le second {`s`, `_`, `e`} totalise $3 + 4 + 4 = 11$ occurrences : les deux totaux sont égaux, on ne peut pas faire plus proche. C'est la séparation de la Figure 2 : le premier groupe part à gauche (branche `1`), le second à droite (branche `0`).
5. La hauteur de l'arbre de la Figure 3 est 5 : la branche la plus longue mène de la racine aux feuilles `o` et `d` en cinq arêtes (racine → [`iucod,pnj`] → [`iucod`] → [`cod`] → [`od`] → `o`). Dans le contexte de l'exercice, la hauteur est la **longueur du plus long code binaire** : les symboles les moins fréquents, `o` et `d`, sont codés sur 5 bits (`11001` et `11000`). C'est le nombre maximal de bits qu'il faut lire pour décoder un symbole.
6. En ASCII, chaque caractère occupe un octet : le texte de 22 caractères occupe 22 octets, soit 176 bits.
Avec le codage de Shannon-Fano, la longueur du code d'un symbole est la profondeur de sa feuille dans la Figure 3 ; on multiplie par le nombre d'occurrences :

symbole	`i`	`u`	`c`	`o`	`d`	`,`	`p`	`n`	`j`	`s`	`_`	`e`
occurrences	1	1	1	1	1	1	1	2	2	3	4	4
bits par symbole	4	4	4	5	5	4	4	4	4	3	3	2
total	4	4	4	5	5	4	4	8	8	9	12	8

Total : $4 + 4 + 4 + 5 + 5 + 4 + 4 + 8 + 8 + 9 + 12 + 8 = 75$ bits, soit 10 octets (9 octets ne font que 72 bits ; il en faut un dixième pour les 3 bits restants). On passe de 22 octets à 10 octets, soit un rapport $22/10 = 2{,}2$: environ deux fois moins d'octets. Le gain vient de ce que les symboles fréquents (`e`, `_`, `s`) ont les codes les plus courts.
7. On applique l'algorithme au mot `chiffrer` (8 lettres).
Étape 1 : occurrences `c` : 1, `h` : 1, `i` : 1, `e` : 1, `f` : 2, `r` : 2 (total 8) ; on les range par nombre croissant : `c`, `h`, `i`, `e`, `f`, `r` (l'ordre entre symboles de même nombre d'occurrences est libre).
Étape 2 : la moitié de 8 est 4 ; les quatre premiers symboles cumulent 4 occurrences, les deux derniers aussi : premier groupe [`c h i e`] (branche `1`), second groupe [`f r`] (branche `0`).
Étape 4, récursivement : [`c h i e`] (total 4, moitié 2) se sépare en [`c h`] et [`i e`], puis chacun en deux feuilles ; [`f r`] (total 4, moitié 2) se sépare en `f` et `r`.



Les codes obtenus sont `c` : `111`, `h` : `110`, `i` : `101`, `e` : `100`, `f` : `01`, `r` : `00` ; le mot `chiffrer` s'encode

en $4 \times 3 + 4 \times 2 = 20$ bits au lieu de 64 en ASCII. Tout arbre de même forme obtenu avec un autre ordre des symboles à une occurrence (par exemple `e i h c`) est également correct.

Partie C **8.** Si le symbole est déjà une clé du dictionnaire, on incrémente son compteur ; sinon on crée la clé avec la valeur 1 :

```
def creer_dico_occ(texte):
    """renvoie un dictionnaire dont les clés sont les
    symboles de texte et les valeurs associées leur
    nombre d'occurences dans texte"""
    dico = {}
    for symbole in texte:
        if symbole in dico:
            dico[symbole] = dico[symbole] + 1
        else:
            dico[symbole] = 1
    return dico
```

Sur `'je pense, donc je suis'`, la fonction renvoie `{'j': 2, 'e': 4, ' ': 4, 'p': 1, 'n': 2, 's': 3, ',': 1, 'd': 1, 'o': 1, 'c': 1, 'u': 1, 'i': 1}`, en accord avec le tableau de la Partie B. Le test d'appartenance `symbole in dico` porte sur les clés et coûte un temps constant en moyenne (table de hachage) : le comptage est linéaire en la longueur du texte.

9. On parcourt le tableau et l'on accumule la seconde composante de chaque tuple :

```
def somme_occ(tab):
    """renvoie la somme des nombres d'occurrences
    des tuples (symbole, nb_occ) du tableau tab"""
    somme = 0
    for couple in tab:
        somme = somme + couple[1]
    return somme
```

Sur le tableau trié de l'exemple, `somme_occ` renvoie `22` ; sur un tableau vide elle renvoie `0`. Sa complexité est linéaire en la longueur du tableau.

10. Après la séparation, si le symbole est dans le premier sous-tableau `t1`, son code commence par `1` et se poursuit par son code *dans `t1`* ; sinon il commence par `0` et se poursuit par son code dans `t2` :

```
def shannon(symbole, tab):
    """renvoie l'écriture binaire associée à symbole
    dans le tableau trié tab"""
    if len(tab) == 1:
        return ""
    else:
        t1, t2 = separe(tab)
        if symbole in [elt[0] for elt in t1]:
            return "1" + shannon(symbole, t1)
        else:
            return "0" + shannon(symbole, t2)
```

Trace de `shannon('s', tab)` sur le tableau trié de l'exemple : `'s'` est dans `t2` = [s, _, e] donc on renvoie `"0" + shannon('s', t2)` ; dans `t2` (total 11, moitié 5), `separe` donne [s, _] et [e], `'s'` est dans le premier : `"1" + shannon('s', [s, _])` ; enfin [s, _] (total 7, moitié 3)

se sépare en [s] et [_], d'où `"1" + shannon('s', [s])` et le cas de base renvoie `""`. Résultat : `"011"`, le code de `s` dans la Figure 3.

11. La fonction est récursive et chaque appel récursif porte sur `t1` ou `t2`, deux sous-tableaux *strictement plus courts* que `tab` : dès que `tab` contient au moins deux symboles, `separe` place au moins un symbole dans chaque partie (la boucle `while` fait au moins un tour car `moitie` $\geq 1 >$ `somme` $= 0$, et s'arrête avant d'avoir tout pris puisque les sommes sont les plus proches possible). La longueur `len(tab)` est donc un **variant** : un entier positif qui décroît strictement à chaque appel. Elle finit par atteindre 1, qui est le **cas de base** (`len(tab) == 1`, ligne 4) : la fonction renvoie `""` sans appel récursif. La récursion s'arrête donc après au plus `len(tab)` -1 appels.

Ce que le correcteur attend : nommer les deux ingrédients, le cas de base et la décroissance stricte de la taille du tableau à chaque appel. On note au passage que la terminaison suppose que `symbole` figure bien dans `tab` et que les deux sous-tableaux sont non vides ; avec le code de `separe` donné, un symbole qui à lui seul dépasse la moitié des occurrences (texte `'aaab'` par exemple) laisse `t2` vide et la fonction ne termine plus (`RecursionError`) : c'est pour cela que le sujet prend soin de *supposer* que `separe` équilibre les deux parties.

12. On construit une seule fois le tableau trié, puis on concatène le code de chaque symbole du texte, dans l'ordre :

```
def encode_shannon(texte):
    """renvoie le mot binaire codant texte
    par l'algorithme de Shannon-Fano"""
    dico = creer_dico_occ(texte)
    tab = creer_tab_trie(dico)
    mot = ""
    for symbole in texte:
        mot = mot + shannon(symbole, tab)
    return mot
```

Testée sur `'je pense, donc je suis'`, la fonction renvoie un mot de 75 bits, la longueur calculée à la question 6. Sur `'chiffrer'` elle renvoie un mot de 20 bits au lieu des 64 du codage ASCII. Le sujet ne dit pas comment `creer_tab_trie` départage les symboles de même nombre d'occurrences, et ce choix change le mot obtenu : avec le tableau `[('e', 1), ('i', 1), ('h', 1), ('c', 1), ('r', 2), ('f', 2)]` (l'ordre qu'on obtient en départageant les ex æquo comme dans l'exemple du sujet) on lit `'10010111000000111101'` ; avec `[('c', 1), ('h', 1), ('i', 1), ('e', 1), ('f', 2), ('r', 2)]`, l'ordre de l'arbre dessiné à la question 7, on lit `'11111010101010010000'`. L'ordre choisi entre ex æquo change les codes, jamais la longueur totale (20 bits dans les deux cas). Sur un texte vide elle renvoie la chaîne vide, et sur un texte formé d'un seul symbole répété (`'aaa'`) elle renvoie aussi la chaîne vide : le tableau trié n'a qu'un élément et son code est vide.

Ce que le correcteur attend : le tableau trié est calculé *avant* la boucle, pas à chaque symbole, et c'est `shannon(symbole, tab)` qui est appelée sur le tableau *complet* (la descente dans les sous-tableaux est faite par la récursion). Remarque : pour le groupe [`,` `p` `n` `j`] (occurrences $1, 1, 2, 2$), deux séparations sont aussi équilibrées ([`,` `p`] | [`n` `j`] et [`,` `p` `n`] | [`j`], écart 2 dans les deux cas) ; le code de `separe` choisit la seconde, si bien que `encode_shannon` attribue `100` à `j`, `1010` à `n`, `10110` à `p` et `10111` à la virgule, codes différents de la Figure 3 mais de même longueur totale (75 bits) : c'est pourquoi le sujet dit qu'on « peut » obtenir l'arbre de la Figure 3.

Exercice 2 (6 points)

Sujet — Exercice 2 — 6 points

Cet exercice porte sur les bases de données relationnelles, le langage SQL et la programmation.
Une ludothèque municipale a décidé de moderniser sa gestion en créant une base de données informatique. Cette base de données permettra de suivre les jeux disponibles, les emprunts effectués par les adhérents, ainsi que les avis laissés sur les différents jeux. Pour commencer, quatre tables principales ont été identifiées : `jeu`, `adhérent`, `emprunt` et `avis`. Ces tables et leurs relations vont permettre de stocker toutes les informations essentielles au bon fonctionnement de la ludothèque. On va considérer que la ludothèque n'a **qu'un exemplaire** de chaque jeu (deux jeux de la ludothèque ne peuvent donc pas avoir le même nom).

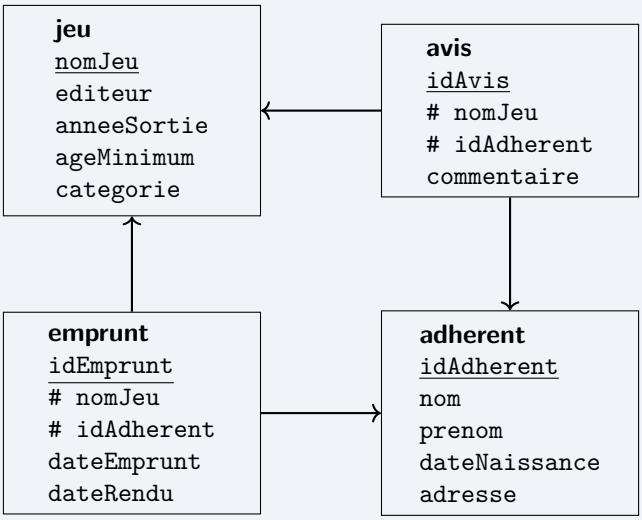


Figure 1. La base de données de la ludothèque

Dans la figure ci-dessus, les clés primaires de chacune des tables sont soulignées et les clés étrangères sont précédées du symbole `#`.
Dans cet exercice, 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` ;
- affiner les recherches à l'aide de `DISTINCT` et `ORDER BY` ;
- réaliser des agrégations à l'aide de `COUNT`.

Par exemple, l'instruction SQL :

```
SELECT COUNT(nomJeu) FROM jeu;
```

donne le nombre de jeux présents dans la table `jeu`.
1. Expliquer pourquoi on ne peut pas prendre l'attribut `nom` comme clé primaire pour la relation `adherent`.
2. Décrire ce que donne la requête SQL suivante :

```
SELECT nomJeu, editeur
FROM jeu
ORDER BY nomJeu;
```

Lorsque qu'un jeu est emprunté et n'a pas encore été rendu, la valeur de l'attribut `dateRendu` de la table `emprunt` est à `NULL`.
3. Écrire une requête permettant de connaitre le nom de tous les jeux qui sont en cours d'emprunt.
4. Écrire une requête SQL pour afficher le nom et le prénom de tous les adhérents qui ont emprunté le jeu « Catan ».
5. Claire VOYANT, adhérente de longue date à cette ludothèque, a emprunté le jeu « Catan » et l'a rendu le 3 juin 2025. Lors de l'emprunt, la valeur de `id_emprunt` était 1538.
Écrire une requête SQL qui a permis de mettre à jour la base de données afin qu'elle prenne en compte que ce jeu a été rendu. Toutes les dates de la base de données sont écrites sous le format `'AAAA-MM-JJ'`.
6. Écrire une requête SQL qui permet de trouver le nom et la catégorie de tous les jeux de la ludothèque sortis à partir de 2010 et dont l'âge minimum est strictement inférieur à 10 ans.
La ludothèque décide d'organiser des événements. Pour cela, elle ajoute une relation `evenement` à sa base de données. En outre, pour chaque événement, elle souhaite garder en mémoire une trace des adhérents qui y ont participé. À cette fin, elle complète sa base avec une relation `participation`.

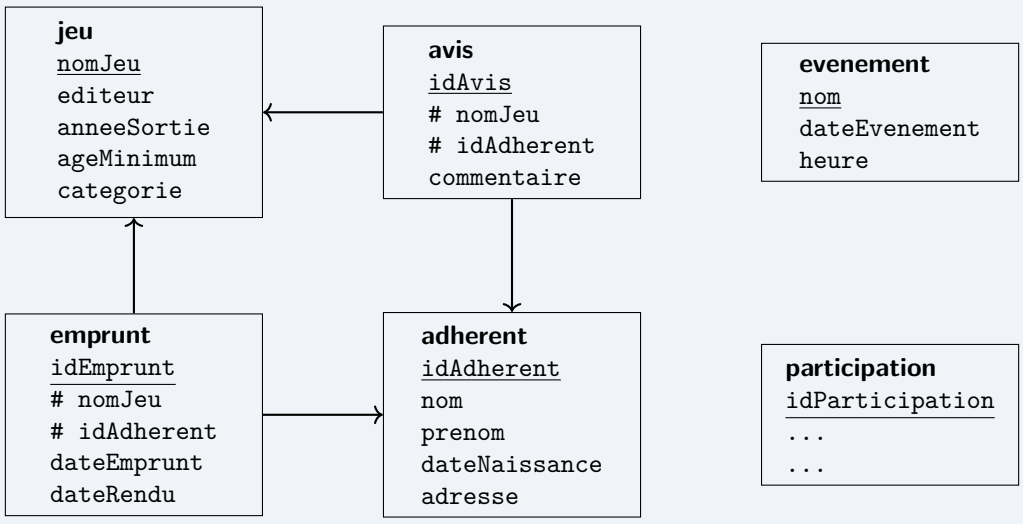


Figure 2. La base de données de la ludothèque actualisée

7. Proposer les clés étrangères de la table `participation` en précisant le nom des attributs auxquels elles font référence.
Le programme Python suivant permet de créer la liste de tous les jeux empruntés, sachant que, dans celle-ci, un jeu va apparaître autant de fois qu'il a été emprunté.

```
1 import sqlite3
2
3 # Connexion à la base de données
```

```
connection = sqlite3.connect("ludotheque.db")
curseur = connection.cursor()

# Exécution de la requête
curseur.execute("SELECT nomJeu FROM emprunt")

# Récupération des résultats
jeux = curseur.fetchall()

liste = []
# Création de la liste des jeux empruntés
for jeu in jeux:
    liste.append(jeu[0])

# Fermeture de la connexion
curseur.close()
connection.close()
```

8. Écrire un script Python permettant de créer le dictionnaire `dict_emprunts` qui, à chaque jeu emprunté, associe le nombre de fois où il a été emprunté.
On veut créer un podium des jeux les plus souvent empruntés. Comme il peut y avoir des égalités à la première, deuxième ou troisième place, il peut y avoir plus de trois jeux sélectionnés sur le podium.
Par exemple, si le dictionnaire des emprunts est :

```
dict_emprunts = {
    "Terraforming Mars": 25,
    "Codenames": 22,
    "Agricola": 18,
    "Puerto Rico": 18,
    "Caylus": 18,
    "Dominion": 22,
    "Dixit": 12
}
```

il y aura sur le podium les jeux « Agricola », « Puerto Rico » et « Caylus » puis les jeux « Dominion » et « Codenames » et enfin le jeu « Terraforming Mars ».
Pour modéliser ce podium en Python, on va utiliser une liste de trois listes.
Pour l'exemple précédent, cette liste sera :

```
[["Agricola", "Puerto Rico", "Caylus"], ["Dominion",
"Codenames"], ["Terraforming Mars"]].
```

9. Proposer un script Python permettant de générer ce podium.

Corrigé

1. Une clé primaire doit identifier de façon **unique** chaque n-uplet de la relation : deux lignes ne peuvent jamais porter la même valeur de clé, et cette valeur ne peut pas être `NULL` (contrainte d'intégrité d'entité). Or rien n'empêche deux adhérents d'être homonymes : deux personnes nommées « VOYANT » (voire deux « Claire VOYANT ») peuvent être inscrites à la ludothèque, et l'attribut `nom` ne permettrait pas de les distinguer, par exemple pour savoir laquelle a emprunté « Catan ». Un nom peut de plus changer au cours du temps. On utilise

donc un identifiant artificiel, `idAdherent`, entier unique attribué à l'inscription et qui ne change jamais.

2. La requête projette deux colonnes de la table `jeu` : elle renvoie, pour *chaque* jeu de la ludothèque (il n'y a pas de clause `WHERE`), son nom et son éditeur, une ligne par jeu, les lignes étant **triées par ordre alphabétique croissant du nom du jeu** (`ORDER BY nomJeu`, l'ordre croissant étant celui par défaut). Comme `nomJeu` est la clé primaire, chaque jeu apparaît exactement une fois ; les jeux jamais empruntés y figurent aussi. Le sujet ne donne le contenu d'aucune table ; les résultats montrés ici et aux questions 4, 8 et 9 ont été obtenus sur une petite base `sqlite3` *construite pour la démonstration* d'après le schéma de la Figure 1 (quatre jeux, trois adhérents, sept emprunts). Sur cette base, le résultat a la forme :

nomJeu	**editeur**
Azul	Next Move
Catan	Kosmos
Codenames	Czech Games
Dixit	Libellud

3. Un jeu est en cours d'emprunt si sa ligne dans `emprunt` n'a pas encore de date de retour :

```sql
SELECT nomJeu
FROM emprunt
WHERE dateRendu IS NULL;
```

Il est inutile de joindre la table `jeu` : le nom du jeu est déjà dans `emprunt` (clé étrangère `nomJeu`).

Ce que le correcteur attend : le test d'une valeur indéfinie s'écrit `IS NULL`, jamais `= NULL` : en SQL, la comparaison `dateRendu = NULL` n'est ni vraie ni fausse et ne sélectionne aucune ligne.

4. Le nom et le prénom sont dans `adherent`, le jeu emprunté dans `emprunt` : il faut une jointure sur la clé étrangère `idAdherent` :

```sql
SELECT DISTINCT adherent.nom, adherent.prenom
FROM adherent
JOIN emprunt ON emprunt.idAdherent = adherent.idAdherent
WHERE emprunt.nomJeu = 'Catan';
```

La jointure associe chaque emprunt à l'adhérent qui l'a réalisé (la condition `ON` égalise la clé étrangère et la clé primaire), puis `WHERE` ne garde que les emprunts de « Catan ». Le mot-clé `DISTINCT` évite d'afficher plusieurs fois un adhérent qui aurait emprunté « Catan » à plusieurs reprises ; sans lui, la requête est acceptée mais Jean DUPONT, qui l'a emprunté deux fois dans la base de test, apparaît deux fois. Une jointure supplémentaire avec `jeu` n'est pas nécessaire, le nom du jeu étant déjà dans `emprunt`.

5. On modifie la ligne de l'emprunt concerné, repérée par sa clé primaire, en renseignant la date de retour au format `'AAAA-MM-JJ'` :

```sql
UPDATE emprunt
SET dateRendu = '2025-06-03'
WHERE idEmprunt = 1538;
```

Le sujet nomme cet attribut `id_emprunt` dans la question ; dans le schéma de la Figure 1 il s'appelle `idEmprunt`, c'est ce nom qui figure dans la base. Comme la clé primaire identifie une ligne unique, il est inutile de préciser le jeu ou l'adhérente.

Ce que le correcteur attend : la clause `WHERE` est indispensable : sans elle, `UPDATE` affecterait la date du 3 juin 2025 à *tous* les emprunts de la table, y compris ceux en cours.

6. Deux conditions sur la table `jeu`, combinées par `AND` : « sortis à partir de 2010 » se traduit par `anneeSortie >= 2010` (2010 inclus) et « âge minimum strictement inférieur à 10 ans » par `ageMinimum < 10` :

```sql
SELECT nomJeu, categorie
FROM jeu
WHERE anneeSortie >= 2010 AND ageMinimum < 10;
```

7. La table `participation` relie un événement à un adhérent : chaque ligne signifie « cet adhérent a participé à cet événement ». Outre sa clé primaire `idParticipation`, elle contient donc deux clés étrangères :

— `#nomEvenement`, qui fait référence à l'attribut `nom`, clé primaire de la relation `evenement` ;

— `#idAdherent`, qui fait référence à l'attribut `idAdherent`, clé primaire de la relation `adherent`.

Schéma : `participation(idParticipation, #nomEvenement, #idAdherent)`. La contrainte d'intégrité référentielle impose que chaque valeur de `nomEvenement` existe dans `evenement.nom` et chaque `idAdherent` dans `adherent` : on ne peut pas enregistrer la participation d'un adhérent inconnu ou à un événement inexistant. Comme un même adhérent peut participer à plusieurs événements et un événement réunir plusieurs adhérents, cette table de liaison est la façon relationnelle de représenter une association « plusieurs à plusieurs ».

8. On reprend le programme du sujet : la requête renvoie une ligne par emprunt, sous forme de tuple `(nomJeu,)` ; il suffit de compter les occurrences de chaque nom dans un dictionnaire, comme à la question 8 de l'exercice précédent :

```python
import sqlite3

# Connexion à la base de données
connection = sqlite3.connect("ludotheque.db")
curseur = connection.cursor()

# Exécution de la requête : une ligne par emprunt
curseur.execute("SELECT nomJeu FROM emprunt")
jeux = curseur.fetchall()

# Comptage des emprunts de chaque jeu
dict_emprunts = {}
for jeu in jeux:
    nom = jeu[0]
    if nom in dict_emprunts:
        dict_emprunts[nom] = dict_emprunts[nom] + 1
    else:
        dict_emprunts[nom] = 1

# Fermeture de la connexion
curseur.close()
connection.close()
```

Exécuté sur une base de test contenant sept emprunts (trois de « Catan », trois de « Dixit », un d'« Azul »), le script construit `{'Catan': 3, 'Dixit': 3, 'Azul': 1}`. Un jeu jamais emprunté n'apparaît pas dans le dictionnaire, conformément à l'énoncé (« à chaque jeu *emprunté* »). On peut aussi partir de la liste `liste` déjà construite par le programme du sujet et écrire `for nom in liste:` avec le même corps de boucle.

9. Le podium est défini par les *valeurs* du dictionnaire : la première place revient à tous les jeux ayant le plus grand nombre d'emprunts, la deuxième à ceux ayant le deuxième plus grand nombre *distinct*, etc. On détermine donc d'abord les nombres d'emprunts distincts, triés par ordre décroissant, puis on range chaque jeu sur la marche qui correspond à sa valeur. Dans l'exemple du sujet, la liste attendue commence par la troisième place et finit par la première : `podium[0]` est la troisième marche, `podium[2]` la première.

```
# 1. Les nombres d'emprunts distincts, du plus grand au plus petit
valeurs = []
for nb in dict_emprunts.values():
    if nb not in valeurs:
        valeurs.append(nb)
valeurs.sort(reverse=True)        # exemple : [25, 22, 18, 12]

# 2. Les trois marches : podium[2] = 1re place, podium[1] = 2e,
#    podium[0] = 3e (ordre de la liste donnée dans le sujet)
podium = [[], [], []]
for jeu in dict_emprunts:
    nb = dict_emprunts[jeu]
    for place in range(3):        # 0 : 1re, 1 : 2e, 2 : 3e place
        if place < len(valeurs) and nb == valeurs[place]:
            podium[2 - place].append(jeu)

print(podium)
```

Sur le dictionnaire du sujet, `valeurs` vaut `[25, 22, 18, 12]` ; les jeux à 18 emprunts vont dans `podium[0]`, ceux à 22 dans `podium[1]`, celui à 25 dans `podium[2]`, et le script affiche `[['Agricola', 'Puerto Rico', 'Caylus'], ['Codenames', 'Dominion'], ['Terraforming Mars']]` : c'est le podium attendu (l'ordre des jeux *à l'intérieur* d'une marche suit l'ordre du dictionnaire et n'a pas d'importance, ce sont des ex æquo). « Dixit », quatrième valeur, est bien écarté. Le test `place < len(valeurs)` évite une erreur d'indice si la ludothèque compte moins de trois nombres d'emprunts distincts : avec `{'Catan': 3, 'Dixit': 3, 'Azul': 1}` le script renvoie `[[], ['Azul'], ['Catan', 'Dixit']]`, et `[[], [], []]` sur un dictionnaire vide.
Ce que le correcteur attend : un tri des valeurs *distinctes* ; trier les jeux par nombre d'emprunts et prendre les trois premiers oublierait les ex æquo (ici six jeux occupent les trois places). Une version qui range `podium[0]` en première place est acceptée si elle le dit explicitement, mais la liste donnée dans le sujet va de la troisième place à la première.

Exercice 3 (8 points)

Sujet — Exercice 3 — 8 points

Cet exercice porte sur la programmation de base en Python, la sécurisation des communications et les réseaux.

Partie A – La méthode du *masque jetable* Dans cette partie, on s'intéresse à une méthode de chiffrement dite du *masque jetable*. Voici ce que l'on peut lire sur le site Wikipédia : *Le chiffrement par la méthode du masque jetable consiste à combiner le message en clair avec une clé présentant les caractéristiques très particulières suivantes :*

- *la clé doit être une suite de caractères au moins aussi longue que le message à chiffrer ;*
- *les caractères composant la clé doivent être choisis de façon totalement aléatoire ;*
- *chaque clé, ou « masque », ne doit être utilisée qu'une seule fois (d'où le nom de masque jetable).*

Illustrons cette méthode par un exemple : on souhaite chiffrer le message ***HELLO*** *avec la clé aléatoire, ou « masque »,* ***WMCKL****.*
Pour cela, on attribue un nombre à chaque lettre, par exemple le rang dans l'alphabet, de 0 à 25.

Tableau de correspondance

Lettre	A	B	C	D	E	F	G	H	I	J	K	L	M
Rang	0	1	2	3	4	5	6	7	8	9	10	11	12

Tableau de correspondance

Lettre	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Rang	13	14	15	16	17	18	19	20	21	22	23	24	25

Ensuite, on additionne la valeur du rang de chaque lettre du message avec la valeur du rang correspondante dans le masque.
Enfin, si le résultat est supérieur à 25 on soustrait 26 (calcul dit « modulo 26 »).
Ainsi, le chiffrement du message ***HELLO*** *avec la clé* ***WMCKL*** *donne le message chiffré* ***DQNVZ*** *comme le montre l'illustration suivante.*

```
     7 (H)   4 (E)  11 (L)  11 (L)  14 (O) message
+   22 (W)  12 (M)   2 (C)  10 (K)  11 (L) masque
=   29      16       13      21      25     masque + message
=    3 (D)  16 (Q)  13 (N)  21 (V)  25 (Z) masque + message  modulo 26
```

Figure 1. Exemple de chiffrement par la méthode du masque jetable
Source : d'après l'article Masque jetable de Wikipédia en français (https://fr.wikipedia.org/wiki/Masque_jetable)

Dans cet exercice, on ne travaillera que sur des chaînes de caractères écrites en majuscules non accentuées (les 26 caractères allant de `'A'` à `'Z'`).

1. Chiffrer, par la méthode du *masque jetable*, le message **LIBRE** à l'aide de la clé **EYQMT**.

En Python, on crée une fois pour toute la variable `alphabet` qui sera accessible et utilisable dans toutes les fonctions. Celle-ci contient la liste des 26 lettres de l'alphabet rangées dans l'ordre alphabétique :

```
alphabet = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J',
'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V',
'W', 'X', 'Y', 'Z']
```

2. Écrire une fonction Python `indice` qui prend pour paramètre une liste `L` et renvoie l'indice de `element` dans la liste `L`.

On supposera que chaque élément de la liste `L` n'y apparaît qu'une seule fois et que `element` est bien présent dans la liste `L`.

Par exemple, l'appel `indice(alphabet, 'K')` renvoie l'entier `10`.

3. Écrire une fonction Python `lettres_vers_indices` qui prend pour paramètre une chaîne de caractères et renvoie, dans l'ordre, la liste des indices de ces caractères dans l'alphabet.

Par exemple, l'appel `lettres_vers_indices('HELLO')` renvoie la liste d'entiers `[7, 4, 11, 11, 14]`.

On dispose également d'une fonction `indices_vers_lettres`, qu'on ne demande pas d'écrire, permettant de convertir une liste d'entiers, compris entre 0 et 25, en une chaîne de caractères.

Par exemple, l'appel `indices_vers_lettres([3, 16, 13, 21, 25])` renvoie la chaîne de caractères `'DQNVZ'`.

Ci-après, on donne une fonction Python `chiffrement` incomplète, qui, à partir d'un message `msg` et d'une clé `cle` entrés en paramètres, renvoie la chaîne de caractères représentant le message chiffré par la méthode du *masque jetable*.

```
1  def chiffrement(msg, cle):
2      assert len(cle) >= len(msg), 'impossible'
3      indices_msg = lettres_vers_indices(msg)
4      indices_cle = lettres_vers_indices(cle)
5      n = len(msg)
6      indices_msg_chiffre = []
7      for k in range(n):
8          ind = ...
9          if ind >= 26:
10             ind = ...
11         indices_msg_chiffre.append(ind)
12     msg_chiffre = indices_vers_lettres(...)
13     return msg_chiffre
```

4. Recopier et compléter les lignes 7 à 13 de la fonction `chiffrement`.

5. Indiquer, en justifiant, ce que l'on observe lors de l'appel `chiffrement('RESEAU', 'GFTZ')`.

On s'intéresse maintenant au déchiffrement d'un message chiffré par la méthode du *masque jetable*.

Par exemple, le déchiffrement du message **DQNVZ** avec la clé **WMCKL** donne le message **HELLO**.

6. Déchiffrer le message **GMEDH** avec la clé **FVEIT**.

7. Expliquer comment procéder pour déchiffrer un message lorsqu'on connaît la clé.

On souhaite maintenant écrire, en Python, une fonction `dechiffrement` qui permet de déchiffrer un message chiffré par la méthode du *masque jetable*.

Pour cela, on s'inspire de la fonction `chiffrement` dans laquelle les paramètres ainsi que les lignes 2 à 5 sont inchangées. On décide cependant de remplacer, ligne 6, le nom de la variable `indices_msg_chiffre` par le nom plus explicite `indices_msg_dechiffre`.
8. Adapter les lignes 6 à 13 de la fonction `chiffrement` pour obtenir la nouvelle fonction `dechiffrement`.

Partie B – Sécurisation des communications **9.** Expliquer la différence entre un algorithme de chiffrement symétrique et un algorithme de chiffrement asymétrique.
Alice souhaite envoyer un message à Bob par l'intermédiaire d'un réseau informatique en utilisant un algorithme de chiffrement asymétrique.
Pour cela, Bob envoie à Alice sa clé publique. Alice chiffre ensuite le message à l'aide de la clé publique de Bob qu'elle vient de recevoir, puis elle envoie ce message chiffré à Bob.
10. Indiquer comment Bob peut déchiffrer le message que lui envoie Alice.
11. Expliquer comment une tierce personne pourrait se faire passer pour Alice sans que Bob ne s'en aperçoive.
12. Expliquer brièvement le fonctionnement du protocole HTTPS.
13. Expliquer pourquoi, pour sécuriser intégralement les communications sur Internet, on utilise le protocole HTTPS plutôt qu'un chiffrement asymétrique.

Partie C – Réseaux Bob et Marc travaillent pour une petite compagnie d'assurances.
Leurs postes de travail font partie d'un même réseau local géré par l'administratrice système qui dispose du bloc d'adresses IPv4 `192.168.110.0/24`.
La notation `/24` situé à la suite de l'adresse `192.168.110.0` signifie que le masque de sous-réseau du réseau de cette entreprise est `255.255.255.0` : les trois premiers octets d'une adresse IP sur ce réseau permettent donc d'identifier la partie réseau de l'adresse, alors que le dernier octet permet d'identifier la partie hôte et est propre à chaque machine sur le réseau. Ce sous-réseau permet donc d'attribuer 256 adresses IPv4 différentes.
L'administratrice choisit alors d'attribuer, en représentation décimale, l'identifiant 115 pour la partie hôte du poste de travail de Bob et l'identifiant 153 pour celui de Marc.
Depuis son poste de travail, Marc souhaite tester la communication avec celui de Bob. Pour cela, il exécute la commande `ping 192.168.100.115` et obtient l'affichage suivant :

```
--- 192.168.100.115 ping statistics ---

4 packets transmitted, 0 received, 100% packet loss, time
3060ms
```

14. Expliquer l'affichage obtenu et corriger l'erreur de Marc.
Afin d'améliorer les performances et la sécurité du réseau de l'entreprise, l'administratrice système décide de séparer le réseau local en plusieurs sous-réseaux et de les relier entre eux par des routeurs. Pour cela, elle modifie le masque de sous-réseau qui devient `11111111.11111111.11111111.11100000`, donné ici en représentation binaire.
15. Donner la représentation décimale de ce masque de sous-réseau.
Pour obtenir l'adresse IPv4 du sous-réseau auquel appartient une machine, il suffit d'appliquer l'opérateur binaire **ET**, bit à bit, entre le masque de sous-réseau et l'adresse IPv4 de la machine.
Par exemple, prenons le dernier octet de l'adresse IPv4 de Bob dont la représentation binaire est `01110011` : en appliquant bit à bit l'opérateur binaire **ET** entre cet octet et l'octet correspondant dans le masque, on obtient le dernier octet de l'adresse du sous-réseau, soit `01100000`.

```
   1 1 1 0 0 0 0 0 (224)
```

```
ET 0 1 1 1 0 0 1 1 (115)
------------------
   0 1 1 0 0 0 0 0 (96)
```

Le poste de travail de Bob est donc sur le sous-réseau d'adresse `192.168.110.96`.

16. Indiquer le nombre total d'adresses IPv4 pouvant être attribuées sur le sous-réseau d'adresse `192.168.110.96` sur lequel se trouve Bob.

L'administratrice système attribue maintenant l'adresse IPv4 `192.168.110.134` au poste de travail de Zoé, nouvelle employée de la compagnie d'assurances.

17. Donner la représentation binaire du nombre 134.

Depuis son poste de travail, Zoé exécute les deux commandes suivantes :

— **commande n°1** : `ping 192.168.110.115` ;

— **commande n°2** : `ping 192.168.110.153`.

18. Indiquer, en justifiant, laquelle de ces deux commandes a produit l'affichage :

```
4 packets transmitted, 4 received, 0% packet loss, time
3002ms
```

Corrigé

Partie A – La méthode du masque jetable **1.** On additionne rang par rang, puis on retranche 26 dès que la somme dépasse 25 :

message	L (11)	I (8)	B (1)	R (17)	E (4)
masque	E (4)	Y (24)	Q (16)	M (12)	T (19)
somme	15	32	17	29	23
modulo 26	15	6	17	3	23
lettre	P	G	R	D	X

Le message chiffré est **PGRDX**. Seules les colonnes I et R ont nécessité la soustraction ($32 - 26 = 6$, $29 - 26 = 3$).

2. La fonction prend deux paramètres, la liste `L` et l'élément cherché, comme le montre l'appel `indice(alphabet, 'K')`. On parcourt la liste jusqu'à trouver l'élément ; l'hypothèse de présence garantit que la boucle s'arrête :

```
def indice(L, element):
    """renvoie l'indice de element dans la liste L
    (element y est présent, une seule fois)"""
    i = 0
    while L[i] != element:
        i = i + 1
    return i
```

`indice(alphabet, 'K')` renvoie 10, `indice(alphabet, 'A')` renvoie 0 et `indice(alphabet, 'Z')` renvoie 25. Une variante avec `for i in range(len(L)):` et `return i` dès que `L[i] == element` est tout aussi correcte. Dans le pire cas (dernier élément), on fait `len(L)` comparaisons : recherche séquentielle, coût linéaire.

3. On applique `indice` à chaque caractère de la chaîne et l'on ajoute le résultat à une liste, dans l'ordre de lecture :

```
def lettres_vers_indices(chaine):
    """renvoie la liste des indices dans alphabet
```

```
    des caractères de chaine, dans l'ordre"""
    indices = []
    for lettre in chaine:
        indices.append(indice(alphabet, lettre))
    return indices
```

`lettres_vers_indices('HELLO')` renvoie `[7, 4, 11, 11, 14]` ; sur la chaîne vide elle renvoie `[]`. La variable globale `alphabet` est lue directement dans la fonction, comme l'énoncé l'autorise.

4. À chaque position `k`, on additionne l'indice de la lettre du message et celui de la lettre de la clé, on ramène la somme entre 0 et 25, puis on reconvertit la liste des indices chiffrés en chaîne :

```
1  def chiffrement(msg, cle):
2      assert len(cle) >= len(msg), 'impossible'
3      indices_msg = lettres_vers_indices(msg)
4      indices_cle = lettres_vers_indices(cle)
5      n = len(msg)
6      indices_msg_chiffre = []
7      for k in range(n):
8          ind = indices_msg[k] + indices_cle[k]
9          if ind >= 26:
10             ind = ind - 26
11         indices_msg_chiffre.append(ind)
12     msg_chiffre = indices_vers_lettres(indices_msg_chiffre)
13     return msg_chiffre
```

Test : `chiffrement('HELLO', 'WMCKL')` renvoie `'DQNVZ'` (l'exemple de la Figure 1) et `chiffrement('LIBRE', 'EYQMT')` renvoie `'PGRDX'` (question 1). Une seule soustraction suffit à la ligne 10 car la somme de deux rangs est au plus $25 + 25 = 50 < 52$; écrire `ind = ind % 26` serait aussi accepté. Si la clé est plus longue que le message, seuls ses `n` premiers caractères servent, puisque la boucle s'arrête à `n = len(msg)`.

5. Le message `'RESEAU'` compte 6 caractères et la clé `'GFTZ'` seulement 4 : la condition `len(cle) >= len(msg)` de la ligne 2 est fausse. L'instruction `assert` interrompt alors l'exécution en levant une exception `AssertionError` accompagnée du message `'impossible'` ; rien n'est chiffré et la fonction ne renvoie aucune valeur. C'est conforme à la première caractéristique du masque jetable : la clé doit être au moins aussi longue que le message, faute de quoi `indices_cle[k]` n'existerait pas pour `k = 4` et `k = 5`.

6. Déchiffrer, c'est retrancher le rang de la lettre de la clé à celui de la lettre chiffrée, en ajoutant 26 si le résultat est négatif :

chiffré	G (6)	M (12)	E (4)	D (3)	H (7)
masque	F (5)	V (21)	E (4)	I (8)	T (19)
différence	1	−9	0	−5	−12
+26 si négatif	1	17	0	21	14
lettre	B	R	A	V	O

Le message clair est **BRAVO**.

7. Le chiffrement ajoute, modulo 26, le rang de la lettre du masque au rang de la lettre du message ; pour l'annuler, on effectue l'opération inverse : pour chaque position, on **soustrait** le rang de la lettre de la clé au rang de la lettre chiffrée ; si le résultat est strictement négatif, on lui **ajoute 26** pour revenir dans l'intervalle de 0 à 25 ; on convertit enfin les rangs obtenus en lettres. La clé sert donc dans les deux sens : c'est un chiffrement *symétrique*. On vérifie

sur l'exemple du sujet : `DQNVZ` avec `WMCKL` donne $3-22=-19\to 7$ (H), $16-12=4$ (E), $13-2=11$ (L), $21-10=11$ (L), $25-11=14$ (O), soit `HELLO`.

8. Seules changent l'opération (soustraction), le test (résultat négatif) et la correction (+26) :

```
def dechiffrement(msg, cle):
    assert len(cle) >= len(msg), 'impossible'
    indices_msg = lettres_vers_indices(msg)
    indices_cle = lettres_vers_indices(cle)
    n = len(msg)
    indices_msg_dechiffre = []
    for k in range(n):
        ind = indices_msg[k] - indices_cle[k]
        if ind < 0:
            ind = ind + 26
        indices_msg_dechiffre.append(ind)
    msg_dechiffre = indices_vers_lettres(indices_msg_dechiffre)
    return msg_dechiffre
```

Tests : `dechiffrement('DQNVZ', 'WMCKL')` renvoie `'HELLO'`, `dechiffrement('GMEDH', 'FVEIT')` renvoie `'BRAVO'` (question 6), et pour tout message `m` et toute clé `c` assez longue, `dechiffrement(chiffrement(m, c), c)` redonne `m` (vérifié sur 200 couples tirés au hasard). *Ce que le correcteur attend :* la différence de deux rangs est comprise entre -25 et 25 : le test est `ind < 0` et non `ind >= 26`, et l'on *ajoute* 26. Un candidat qui recopie `ind >= 26` sans changer le test a une fonction qui ne corrige jamais rien et renvoie des indices négatifs.

Partie B – Sécurisation des communications **9.** Dans un chiffrement **symétrique**, une *même clé secrète* sert à chiffrer et à déchiffrer (le masque jetable de la Partie A, ou AES) : les deux correspondants doivent posséder cette clé, ce qui pose le problème de son échange sur un canal non sûr. Dans un chiffrement **asymétrique** (RSA), chaque participant possède une *paire de clés* : une clé *publique*, diffusée librement, et une clé *privée*, jamais transmise ; ce qui est chiffré avec la clé publique ne peut être déchiffré qu'avec la clé privée correspondante. Il n'y a plus de secret à échanger, mais les calculs sont beaucoup plus lents.

10. Le message a été chiffré avec la clé publique de Bob : seul le possesseur de la clé **privée** associée peut le déchiffrer. Bob déchiffre donc avec *sa clé privée*, qu'il n'a jamais communiquée à personne ; même un espion qui aurait capté la clé publique et le message chiffré n'en tire rien.

11. La clé publique de Bob est, par définition, publique : elle a circulé en clair sur le réseau et n'importe qui peut l'intercepter ou l'obtenir. Une tierce personne, Ève, peut donc rédiger un message de son choix, le chiffrer avec la clé publique de Bob et l'envoyer en prétendant être Alice (par exemple en signant « Alice » ou en usurpant son adresse). Bob le déchiffre normalement avec sa clé privée et n'a aucun moyen de savoir qui l'a écrit : le chiffrement asymétrique garantit la *confidentialité* (seul Bob peut lire), pas l'*authentification* de l'expéditeur. Pour s'en prémunir, Alice devrait *signer* son message avec sa propre clé privée, signature que Bob vérifierait avec la clé publique d'Alice.

12. HTTPS est le protocole HTTP encapsulé dans le protocole TLS (port 443 au lieu de 80). Lors de la connexion, le serveur envoie au navigateur son *certificat*, qui contient sa clé publique et est signé par une autorité de certification de confiance ; le navigateur vérifie ce certificat (authentification du serveur), génère une *clé de session* symétrique et la transmet au serveur chiffrée avec la clé publique de celui-ci (chiffrement asymétrique) ; la suite des échanges HTTP est ensuite chiffrée avec cette clé de session (chiffrement symétrique, rapide). HTTPS apporte ainsi la confidentialité, l'authentification du serveur et l'intégrité des données (toute altération est détectée).

13. Un chiffrement asymétrique seul présente deux défauts. D'abord il est *lent* : ses calculs sur de grands nombres sont beaucoup plus coûteux que ceux d'un algorithme symétrique, et chiffrer ainsi tout le trafic d'une page web ou d'une vidéo serait prohibitif ; HTTPS ne l'emploie que pour échanger la clé de session, puis chiffre le reste en symétrique. Ensuite, comme l'a montré la question 11, il n'*authentifie* pas l'interlocuteur : rien n'empêche un pirate de se faire passer pour le serveur (ou pour Alice) ; HTTPS règle ce point avec le certificat délivré par une autorité de confiance, et garantit en plus l'intégrité des échanges. HTTPS combine donc le meilleur des deux méthodes : la sécurité de l'échange de clé asymétrique, la vitesse du chiffrement symétrique et l'authentification par certificat.

Partie C – Réseaux **14.** Marc a fait une faute de frappe dans le troisième octet : il a tapé `192.168.100.115` au lieu de `192.168.110.115`. Avec le masque `255.255.255.0`, les trois premiers octets identifient le réseau : l'adresse tapée appartient au réseau `192.168.100.0/24`, qui n'est pas le réseau local de l'entreprise. Aucune machine ne répond à cette adresse (ou les paquets ne sont même pas routés) : les quatre requêtes ICMP envoyées par `ping` restent sans réponse, d'où « 4 packets transmitted, 0 received, 100 % packet loss ». La commande correcte, avec l'identifiant d'hôte 115 de Bob, est `ping 192.168.110.115`.
15. On convertit chaque octet : `11111111` vaut 255 et `11100000` vaut $128 + 64 + 32 = 224$. Le masque s'écrit `255.255.255.224`, soit `/27` (27 bits à 1).
16. Le masque laisse $32 - 27 = 5$ bits à 0 pour la partie hôte : il y a $2^5 = 32$ combinaisons, soit **32 adresses** sur le sous-réseau `192.168.110.96`, de `192.168.110.96` à `192.168.110.127` (c'est la convention du sujet, qui compte 256 adresses pour le `/24`). En pratique deux d'entre elles sont réservées, l'adresse du sous-réseau `.96` (bits hôte tous à 0) et l'adresse de diffusion `.127` (bits hôte tous à 1), si bien que $2^5 - 2 = 30$ machines au plus peuvent recevoir une adresse. Le réseau initial est ainsi découpé en $2^3 = 8$ sous-réseaux de 32 adresses.
17. $134 = 128 + 4 + 2 = 2^7 + 2^2 + 2^1$, d'où l'écriture binaire `10000110` (on vérifie : $128 + 0 + 0 + 0 + 0 + 4 + 2 + 0 = 134$).
18. On calcule le sous-réseau de chaque poste par un **ET** bit à bit du dernier octet avec `11100000` :

poste	dernier octet	binaire	**ET** `11100000`	sous-réseau
Zoé	134	`10000110`	`10000000` = 128	`192.168.110.128`
Bob	115	`01110011`	`01100000` = 96	`192.168.110.96`
Marc	153	`10011001`	`10000000` = 128	`192.168.110.128`

Zoé et Marc sont sur le *même* sous-réseau `192.168.110.128` (adresses de `.128` à `.159`) : les paquets circulent directement entre leurs postes et les quatre réponses reviennent. Bob est sur un autre sous-réseau (`192.168.110.96`), désormais séparé par un routeur, mis en place précisément pour la sécurité : la communication n'est plus directe et peut être filtrée. C'est donc la **commande n°2**, `ping 192.168.110.153` vers le poste de Marc, qui a produit l'affichage « 4 received, 0 % packet loss ».
Ce que le correcteur attend : le calcul du ET pour les *trois* adresses, qui montre que 134 et 153 partagent les trois premiers bits `100` de leur dernier octet alors que 115 commence par `011` ; répondre « la n°2 » sans ce calcul ne justifie rien.