BACCALAURÉAT GÉNÉRAL

ÉPREUVE D'ENSEIGNEMENT DE SPÉCIALITÉ

SESSION 2025

NUMÉRIQUE ET SCIENCES INFORMATIQUES

ÉPREUVE DU MERCREDI 18 JUIN 2025

Durée de l'épreuve : **3 heures 30**

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

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

Ce sujet comporte 15 pages numérotées de 1/15 à 15/15.

Le sujet est composé de trois exercices indépendants.

Le candidat traite les trois exercices.

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.

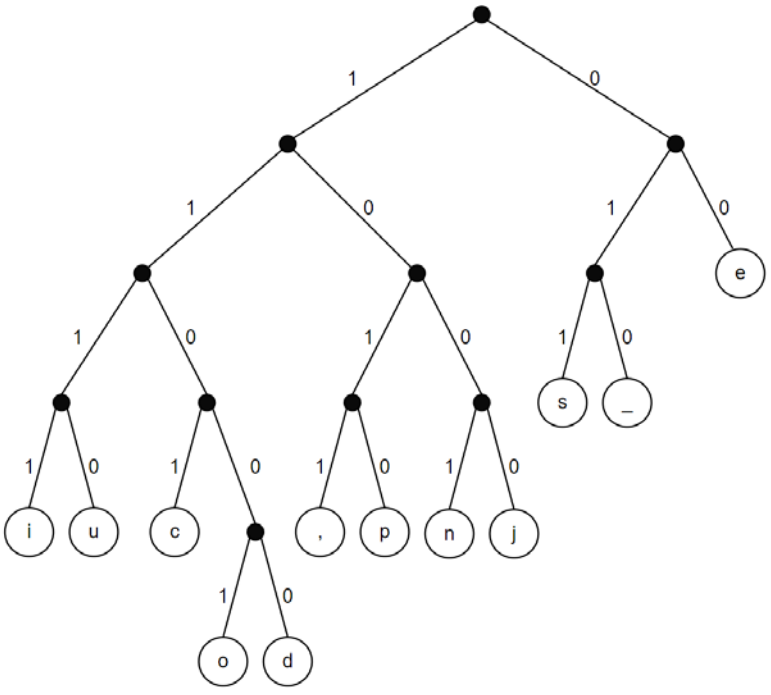


Figure 1. Exemple d'arbre de codage

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 `000111010111111001100 1`.

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.

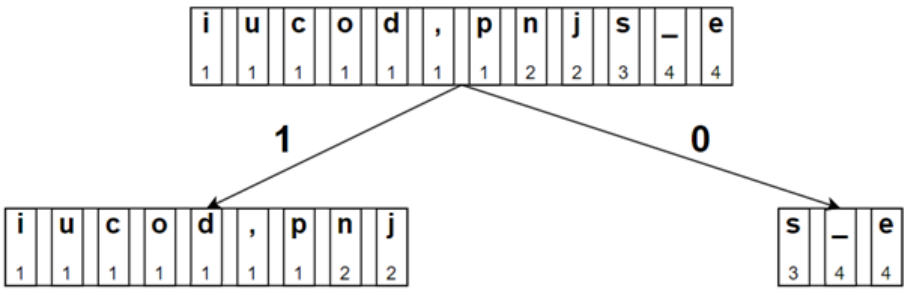


Figure 2. Le résultat de l'étape 2

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

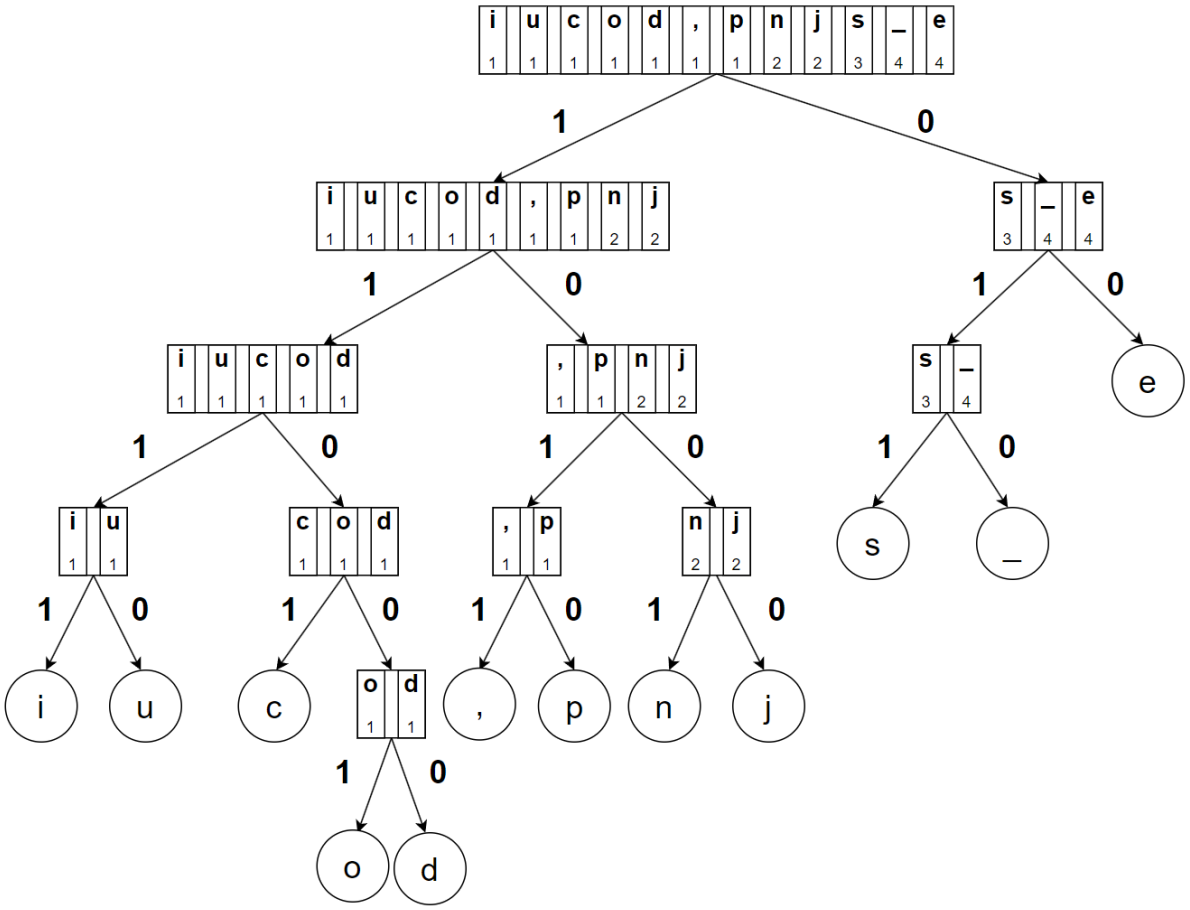


Figure 3. Arbre de codage obtenu par l'algorithme de Shannon-Fano

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:
 5         return ""
 6     else:
 7         t1, t2 = separe(tab)
 8         if symbole in [elt[0] for elt in t1]:
 9             return "1" + ...
10         else:
11             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
```

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).

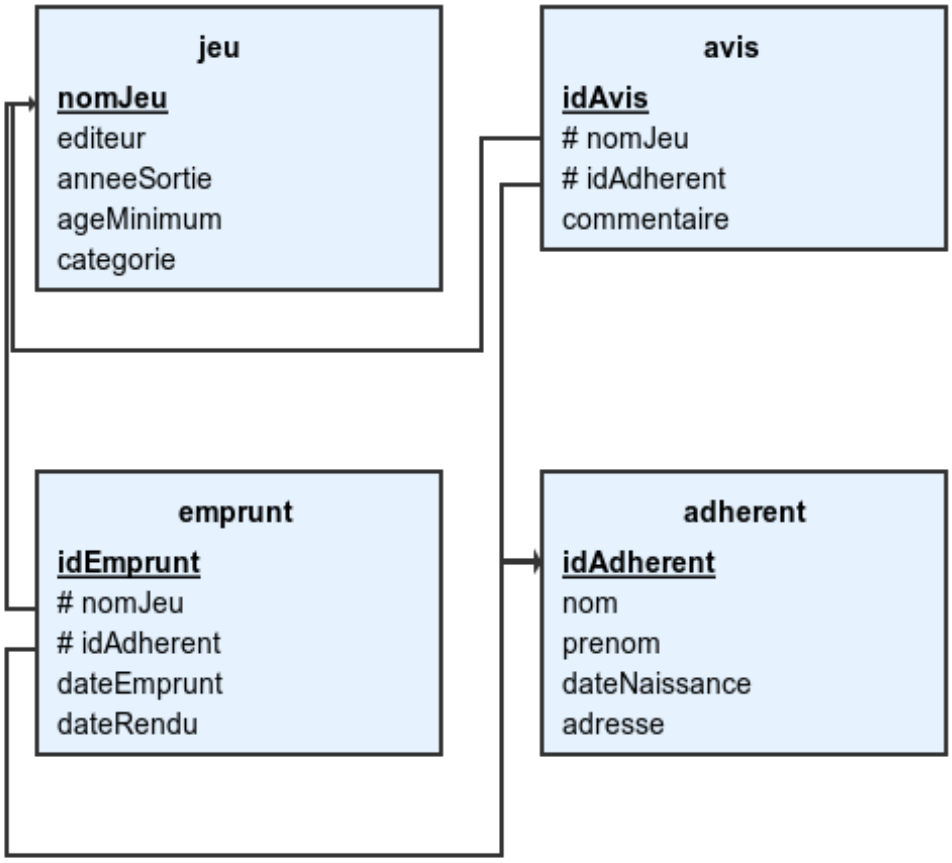


Figure 1. La base de données de la ludothèque

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`.

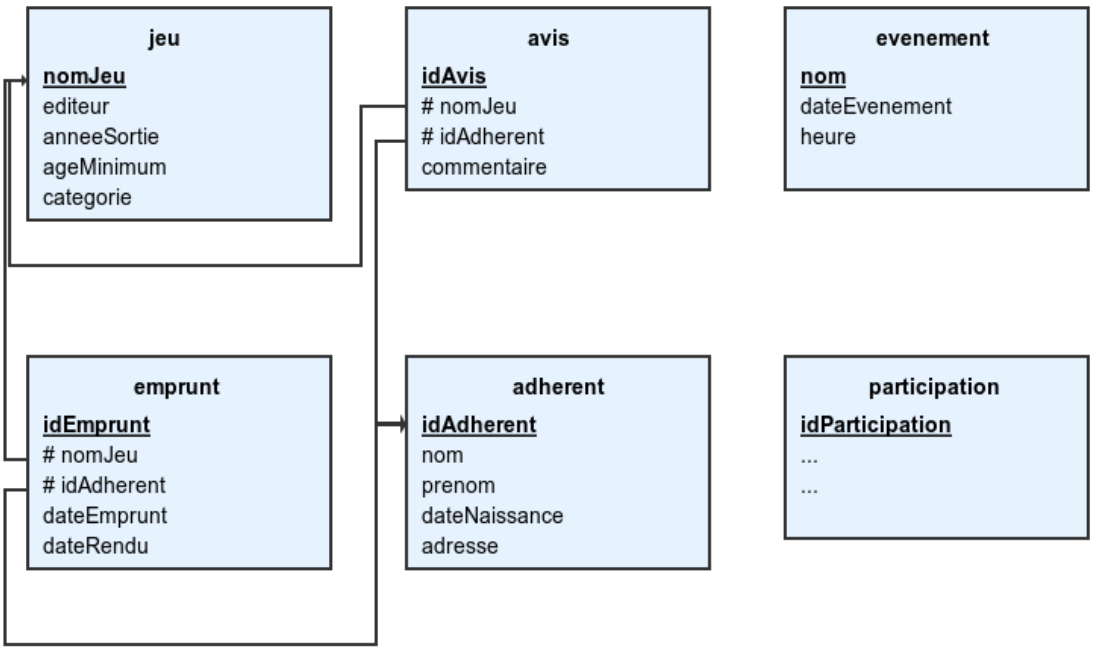


Figure 2. La base de données de la ludothèque actualisée

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.

```
import sqlite3

# 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.

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
```