BACCALAURÉAT GÉNÉRAL

ÉPREUVE D'ENSEIGNEMENT DE SPÉCIALITÉ

SESSION 2026

NUMÉRIQUE ET SCIENCES INFORMATIQUES

ÉPREUVE DU MARDI 16 JUIN 2026

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 13 pages numérotées de 1/13 à 13/13.

Le sujet est composé de trois exercices indépendants.

Le candidat traite les trois exercices.

Exercice 1 (6 points)

Cet exercice porte sur le modèle relationnel, les systèmes de gestion de bases de données relationnelles et le langage SQL.

Le restaurant *Pizzayolo* cherche à moderniser son système de réservation de pizzas et fait appel à un prestataire informatique qui lui propose d'utiliser un système de gestion de bases de données relationnelles.

1. Donner deux avantages que peut apporter un système de gestion de bases de données par rapport à l'utilisation d'un simple tableur dans lequel on ajouterait chaque nouvelle commande sur une ligne.

Le prestataire propose une base de données constituée de trois relations selon le schéma relationnel suivant, les clés primaires des tables étant soulignées.

`client`(`id` : *entier*, `nom` : *texte*, `prenom` : *texte*)

`pizza`(`num` : *entier*, `couleur` : *texte*, `prix` : *flottant*)

`commande`(`code` : *entier*, `id_client` : entier, `num_pizza` : entier, `date` : *texte*, `livraison` : *entier*, `paiement` : *entier*)

Voici un extrait des tables `client`, `pizza` et `commande` :

client		
id	nom	prenom
103	Esposito	Giulia
106	Panzini	Raffaele

pizza		
num	couleur	prix
1	Rossa	11.5
2	Bianca	11.5
3	Nera	12.5
4	Bianca	15.5

commande					
code	id_client	num_pizza	date	livraison	paiement
42362	103	2	2025-01-02	1	1
42363	103	3	2025-01-02	1	1
42364	106	1	2025-01-03	1	0
42365	103	1	2025-01-03	0	1

Chaque pizza commandée par un client donne lieu à une commande différente. Ainsi, un même client peut, par exemple, commander plusieurs pizzas le même jour : ceci se traduit dans la table par plusieurs commandes de codes différents. La pizza d'une commande peut avoir été livrée ou non : l'attribut `livraison` vaut 1 si la pizza a été livrée et 0 sinon. Le client peut avoir payé sa commande ou non : l'attribut `paiement` vaut 1 si la commande a été réglée et 0 sinon.

2. Indiquer, en justifiant, s'il est possible d'avoir deux pizzas avec le même numéro de pizza, le même prix et avec une couleur de pizza différente.

3. Donner un exemple de clé étrangère en précisant la relation à laquelle elle appartient ainsi que l'attribut et la relation qu'elle référence.

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`), `JOIN ... ON` ;
- construire des requêtes d'insertion et de mise à jour à l'aide de `UPDATE`, `INSERT`, `DELETE` ;
- affiner les recherches à l'aide de `DISTINCT`, `ORDER BY`.

Les fonctions d'agrégation `COUNT(attribut)`, `AVG(attribut)`, `MAX(attribut)`, `MIN(attribut)` et `SUM(attribut)` renvoient, respectivement, le nombre, la moyenne, la plus grande, la plus petite et la somme des valeurs de l'attribut sélectionné.

Par exemple, la requête `SELECT MAX(prix) FROM pizza;` renvoie le prix de la pizza la plus chère et `SELECT SUM(livraison) FROM commande;` renvoie le nombre total de pizzas qui ont été livrées.

4. Écrire le résultat renvoyé par la requête SQL suivante, en s'appuyant uniquement sur les extraits des tables fournis précédemment.

```
SELECT prix FROM pizza WHERE couleur = 'Bianca';
```

5. Écrire une requête SQL qui donne les couleurs des pizzas dont le prix est strictement supérieur à 15 €.

6. Écrire une requête SQL qui donne les prénoms des clients qui n'ont pas payé une pizza qui leur a pourtant été livrée.

7. Écrire une requête SQL qui donne le prix moyen de vente des pizzas commandées par un client de nom « Esposito ».

La commande de code `42365` vient tout juste d'être livrée.

8. Écrire une requête SQL qui permet de mettre à jour la base de données en conséquence.

Le 3 janvier 2025, Emanuele Girasole, qui n'a jamais encore commandé dans ce restaurant, souhaite se faire livrer la pizza numéro 2.

9. Écrire, dans le bon ordre, les requêtes SQL qui permettent de mettre à jour la base de données en conséquence. On suppose que l'identifiant client 107 et que le code de commande 42366 n'ont pas encore été attribués dans la base de données.

La gérante de la pizzéria décide de retirer de la vente la pizza d'identifiant 1 et exécute la requête suivante :

```
DELETE FROM pizza
WHERE num = 1;
```

10. Indiquer le problème que peut poser l'exécution de cette requête.

La gérante aimerait que l'on ajoute dans la base de données un moyen de retrouver les ingrédients utilisés pour chaque pizza. Chaque pizza comporte divers ingrédients et un même ingrédient peut être utilisé sur plusieurs pizzas. Il est donc nécessaire de proposer un nouveau schéma relationnel en ajoutant deux relations : une table `ingredient` et une table `composition` qui permet de faire le lien entre les ingrédients et les pizzas.

11. Écrire le schéma relationnel des deux nouvelles tables en indiquant clairement les clés primaires et les clés étrangères.

Afin de pouvoir permettre à chaque client de consulter sur Internet l'historique de ses commandes, le prestataire propose de mettre la base de données sur un serveur accessible par des requêtes extérieures et non uniquement via le réseau interne utilisé par le restaurant.

12. Indiquer les problèmes soulevés par cette démarche et les précautions à prendre pour y remédier.

Exercice 2 (6 points)

Cet exercice porte sur l'algorithmique et la programmation Python.

La recherche d'une plus longue sous-liste commune à deux listes est un problème connu sous le nom de « Longest Common Subsequence » (LCS).

Les éléments d'une plus longue sous-liste commune sont des éléments communs aux deux listes initiales, où ils apparaissent dans le même ordre, mais pas nécessairement de façon contiguë. Ainsi, les listes `L_1 = [9, 3, 7, 5, 8]` et `L_2 = [9, 7, 8, 3, 7, 3]` ont notamment en commun les sous-listes `[]`, `[9]`, `[3]`, `[9, 7]`, `[9, 3, 7]` ou `[9, 7, 8]`. Les plus longues sous-listes communes à ces deux listes sont `[9, 3, 7]` et `[9, 7, 8]`, de longueur commune égale à 3.

En Python, il est possible d'obtenir la concaténée de deux valeurs de types *list* en utilisant +. Par exemple, [1, 2] + [3, 4] s'évalue à [1, 2, 3, 4] et [ ]+[1] s'évalue à [1].

Partie A.

Dans cette partie, on s'attache à déterminer une LCS selon la stratégie de « recherche exhaustive ». La première fonction implémentée construit toutes les sous-listes d'une liste passée en paramètre, en appliquant l'algorithme suivant :

```
Fonction sous_listes(L: list, accumulateur: list[list])
         ->list[list]
Début
|  Si L est vide alors
|  |  renvoyer l'accumulateur (initialement égal à [[]])
|  Sinon
|  |  Extraire e le dernier élément de L
|  |  Créer L_t une liste temporaire vide
|  |  Ajouter dans L_t chaque sous-liste présente dans
|  |  l'accumulateur, après lui avoir ajouté, en tête, e
|  |  Renvoyer le résultat d'un nouvel appel à la fonction
|  |  avec pour paramètres
|  |  - la liste diminuée de cet élément e
|  |  - l'union des éléments de l'accumulateur et de la
|  |  liste temporaire
|  FinSi
Fin
```

1. Recopier et compléter le tableau suivant, en indiquant la valeur des paramètres lors des appel récursifs, pour un appel initial `sous_listes([9, 3, 7], [[]])`.

Évolution des paramètres lors des appels récursifs		
Appel	liste	accumulateur
appel initial	[9, 3, 7]	[[]]
1er sous-appel	[9, 3]	[[], [7]]
2nd sous-appel		
dernier sous-appel		

La fonction suivante implémente l'algorithme étudié précédemment.

```
1 def sous_listes(liste: list, accumulateur = [[]]):
2     if len(liste) ==  0:
3         return accumulateur
4     element = liste.pop()
5     tmp = []
6     for sous_liste in accumulateur:
7         tmp.append(...)
8     return sous_listes(...)
```

2. Recopier et compléter les lignes 7 et 8 du code de la fonction `sous_listes`.

On implémente la fonction `est_extraite` ci-dessous qui renvoie `True` si une sous-liste peut être extraite d'une liste donnée et `False` sinon.

```
def est_extraite(sous_liste: list, liste: list):
    index_ss_liste, index_liste = 0, 0
    while index_ss_liste < len(sous_liste) and index_liste <
len(liste):
        if sous_liste[index_ss_liste] == liste[index_liste]:
            index_ss_liste += 1
        index_liste += 1
    return index_ss_liste == len(sous_liste)
```

On implémente la fonction `lcs_force_brute` ci-dessous qui détermine une plus longue sous-liste commune à deux listes données et qui renvoie cette liste.

```
def lcs_force_brute(liste_1: list, liste_2: list):
    tmp = sous_listes(liste_1)
    resultat, maxi = [], 0
    for element in tmp:
        if est_extraite(element, liste_2) and len(element) >
maxi:
            resultat = element
```

```
            maxi = len(element)
    return resultat
```

3. Expliquer pourquoi l'exécution de la fonction `lcs_force_brute` risque de prendre beaucoup de temps selon la longueur des listes passées en paramètre. Justifier.

Partie B.

Soient m et n deux entiers naturels, m' et n' des entiers tels que $0 \leq m' < m$ et $0 \leq n' < n$. On suppose, dans cette partie, disposer d'une fonction `lcs` qui, pour deux listes quelconques `L_1` et `L_2`, de tailles respectives m et n, et des entiers m' et n' passés en paramètres, renvoie une plus longue sous-liste commune à

- la sous-liste de `L_1` constituée des m' premiers éléments ;
- et la sous-liste de `L_2` constituée des n' premiers éléments.

Ainsi,

- `lcs([9, 3, 7, 5, 8], 0, [9, 7, 8, 3, 7, 3], 0)` est une plus longue sous-liste commune à deux listes vides, soit `[]` ;
- `lcs([9, 3, 7, 5, 8], 3, [9, 7, 8, 3, 7, 3], 2)` est une plus longue sous-liste commune à `[9, 3, 7]` et `[9, 7]`, soit la sous-liste `[9, 7]`.

4. Indiquer ce que renvoie l'appel suivant :

```
lcs([9, 3, 7, 5, 8], 1, [9, 7, 8, 3, 7, 3], 1)
```

On suppose, dans cette question, que `L_1` et `L_2` sont des listes non vides dont les derniers éléments sont identiques : $m \geq 1$, $n \geq 1$, `L_1[m-1]` et `L_2[n-1]` égaux.

Exemple : $m = 5$, $n = 3$, `L_1 = [9, 3, 7, 5, 8]` et `L_2 = [9, 7, 8]`.

5. Écrire la relation de récurrence pour obtenir une plus longue sous-liste commune aux deux listes `L_1` et `L_2` à partir du résultat de `lcs(L_1, m-1, L_2, n-1)`.

A contrario, lorsque les derniers éléments des listes sont différents ($m \geq 1$, $n \geq 1$, `L_1[m-1]` et `L_2[n-1]` distincts), une plus longue sous-suite commune aux deux listes `L_1` et `L_2` s'obtient en choisissant une plus grande liste parmi les résultats de `lcs(L_1, m-1, L_2, n)` et `lcs(L_1, m, L_2, n-1)`.

Par exemple, avec $m = 5$, $n = 4$, `L_1 = [9, 3, 7, 5, 8]` et `L_2 = [9, 7, 8, 3]`, on compare les longueurs d'une LCS de `[9, 3, 7, 5]` et `L_2 = [9, 7, 8, 3]` (soit `[9, 7]`) et d'une LCS de `L_1 = [9, 3, 7, 5, 8]` et `[9, 7, 8]` (soit `[9, 7, 8]`), ce qui permet d'obtenir la LCS `[9, 7, 8]` pour `L_1` et `L_2`.

On implémente donc la fonction suivante qui prend en paramètres deux listes et qui renvoie une plus longue sous-liste commune :

```
 1 def lcs_top_down(liste_x: list, liste_y: list):
 2     def aux(liste_x, index_x, liste_y, index_y):
 3         if (index_x < 0) or (index_y < 0):
 4             return []
 5         if liste_x[index_x] == liste_y[index_y]:
 6             return aux(liste_x, index_x-1, liste_y, index_y-
1) + [liste_x[index_x]]
 7         resultat_1 = aux(liste_x, index_x, liste_y, index_y-
1)
 8         resultat_2 = aux(liste_x, index_x-1, liste_y,
index_y)
 9         if len(resultat_1) >= len(resultat_2):
10             return resultat_1
11         return resultat_2
12     return aux(liste_x, len(liste_x)-1, liste_y,
len(liste_y)-1)
```

6. Justifier que la fonction `aux` contenue dans la fonction `lcs_top_down` est récursive.

7. Détailler le risque d'un tel choix dans la présente situation, lorsque les listes `L_1` et `L_2` sont très grandes.

8. Proposer une solution pour éviter le problème évoqué à la question 7.

Exercice 3 (8 points)

Cet exercice porte sur les protocoles de routage, la sécurisation des communications et la programmation orientée objet.

Bob, un particulier, souhaite communiquer de façon sécurisée avec un serveur web qui héberge un site dont le nom de domaine est *alice.fr*.

Dans tout l'exercice, pour plus de simplicité d'écriture, on appelle *Bob* le navigateur et *Alice* le serveur web.

Partie A

On donne ci-après le réseau entre Alice et Bob. Le coût d'une liaison est marqué à côté de chaque arête.

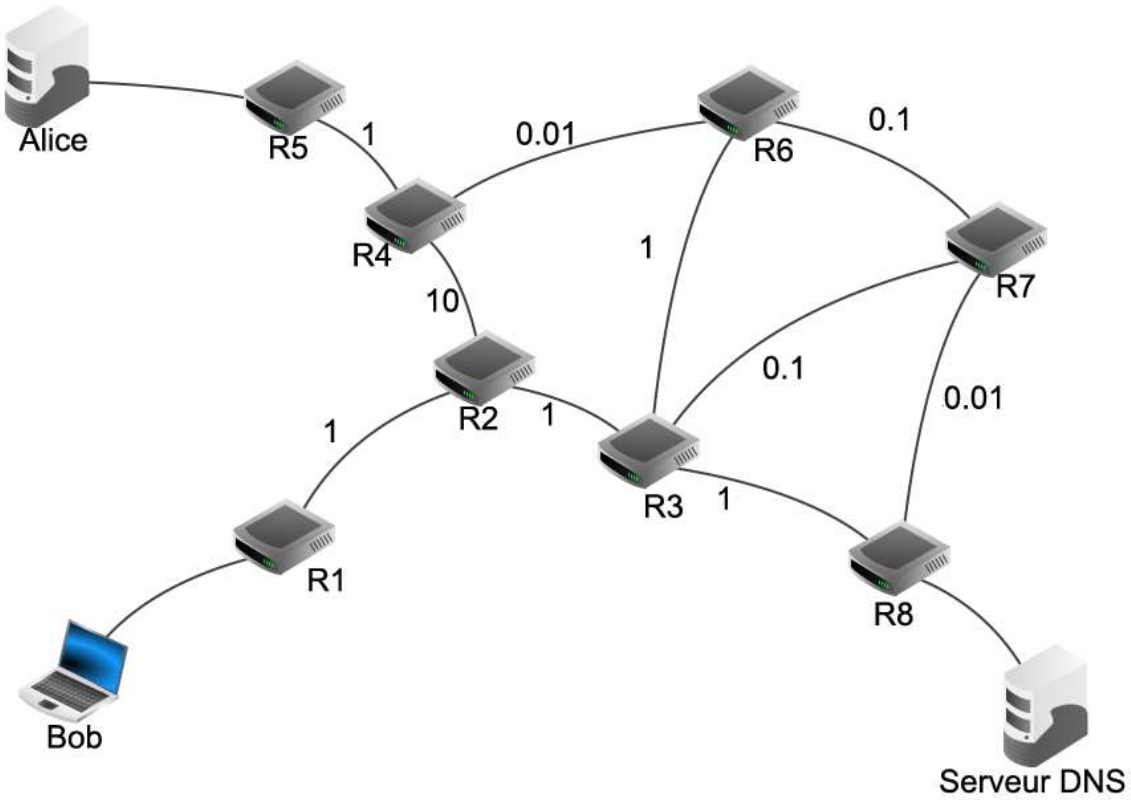

Figure 1. Schéma du réseau

La première fois que l'URL *http://alice.fr* est saisie dans la barre d'adresse de Bob, ce dernier envoie tout d'abord une première requête au serveur DNS afin d'obtenir l'adresse IP d'Alice avant de pouvoir envoyer une requête à Alice.

Le protocole RIP est un protocole de routage qui minimise le nombre de routeurs par lesquels les paquets transitent.

1. Donner, en utilisant le protocole RIP, le chemin que les paquets emprunteront afin qu'une requête de Bob parvienne au serveur DNS. On ne fera apparaître que les routeurs traversés.

Le protocole OSPF est un protocole de routage qui minimise le coût du transit des paquets.

2. On suppose que Bob a l'adresse IP d'Alice. Donner, en utilisant le protocole OSPF, le chemin que les paquets emprunteront afin qu'une requête de Bob parvienne à Alice. On ne fera apparaître que les routeurs traversés.

3. Donner le coût du chemin identifié dans la question 2.

4. Expliquer ce qui se passerait si le routeur R3 tombe en panne. On n'attend pas l'écriture des tables de routage.

5. Indiquer un protocole permettant à un navigateur et un serveur web de communiquer en chiffrant leurs données.

Bob et Alice souhaitent communiquer de façon sécurisée.

Bob génère une clé symétrique qu'il doit transmettre à Alice. Bob et Alice possèdent alors cette même clé symétrique.

6. Expliquer le rôle qu'aura cette clé symétrique dans leur communication.

Bob doit transmettre la clé symétrique qu'il a générée à Alice sans qu'un tiers puisse l'intercepter et l'utiliser pour les espionner.
Pour cela, Alice génère un couple de clés composé d'une clé publique et d'une clé privée.

7. Expliquer comment Alice et Bob vont utiliser cette clé publique et cette clé privée afin que Bob puisse transmettre de façon sécurisée sa clé symétrique à Alice. On ne parlera, dans cette question, ni de signature ni de certificat.

Lorsqu'Alice reçoit la clé symétrique chiffrée censée provenir de Bob, elle veut avoir la certitude que c'est bien Bob qui la lui a envoyée.

Pour cela, Bob utilise un système de signature digitale basé sur un couple (clé privée, clé publique) et dans lequel seule la clé privée permet de chiffrer et la clé publique permet de déchiffrer. On parle alors de signer plutôt que de chiffrer, et de signature plutôt que de chiffrement. Bob procède alors de la façon suivante :

- il envoie sa clé publique à Alice ;
- il crée un résumé, appelé condensé, des données de la clé symétrique qu'il veut transmettre à Alice. Il est alors possible de comparer le résumé et la clé pour voir s'ils correspondent bien l'un à l'autre ;
- il signe ce condensé en utilisant sa propre clé privée ;

- il transmet à Alice l'ensemble composé de la clé symétrique et de la signature, après l'avoir comme précédemment chiffré grâce à la clé publique transmise par Alice.

8. Expliquer pourquoi Alice est la seule à pouvoir accéder à la clé symétrique et peut avoir la certitude que c'est bien Bob qui la lui a envoyée.

Partie B

Dans cette partie, on se propose de coder, dans un module que l'on nommera `outils`, quelques fonctions qui serviront pour la suite de l'exercice.

9. Écrire le code d'une fonction `somme` qui prend en paramètre un tableau (type `list` en Python) d'entiers et renvoie la somme des éléments de ce tableau.

   Exemple :

```
>>> somme([1, 3, 5, 7, 9])
25
```

10. Écrire le code d'une fonction `permuter` qui :

    – prend en paramètres un tableau `tab` (type `list` en Python) et deux entiers `i` et `j` ;
    – modifie le tableau `tab` en permutant ses éléments d'indices `i` et `j`.

    Exemple :

```
>>> liste = [1, 3, 5, 7, 9]
>>> permuter(liste, 1, 3)
>>> liste
[1, 7, 5, 3, 9]
```

11. Recopier et compléter la ligne 3 du code ci-après de la fonction `inverser` qui modifie le tableau passé en paramètre en inversant l'ordre de ses éléments.

```
1 def inverser(tab):
2   for i in range(len(tab) // 2):
3     permuter(...)
```

On appelle `sac` un tableau (type `list` en Python) de 8 éléments entiers construit de la manière suivante.

- Le premier élément du `sac` est un entier choisi au hasard entre 1 et 5 tous deux inclus.
- Pour ajouter un nouvel entier au `sac` :
  – on calcule la somme `S` de tous les éléments du `sac` ;
  – on choisit au hasard un entier entre `S+1` et `2*S` tous deux inclus ;
  – on ajoute cet entier dans le `sac` à la suite des précédents.

- Une fois que le `sac` contient ses huit entiers, on inverse l'ordre de ses éléments afin que le premier élément du sac soit le plus grand.

On rappelle que la fonction `randint(a, b)` du module `random` renvoie un nombre entier choisi aléatoirement entre `a` et `b` tous deux inclus.

12. Recopier et compléter les lignes 7 à 13 du code ci-après de la fonction `generer_sac`.

```
 1 from random import randint
 2 def generer_sac():
 3   '''
 4   renvoie un sac (8 entiers)
 5   : return (list) un sac
 6   '''
 7   sac = ...
 8   for i in range(7):
 9     s = ...
10     nouvel_entier = ...
11     sac.append(nouvel_entier)
12   ...
13   return sac
```

Partie C

Dans cette partie, on code une classe `Cle_symetrique` qui permet d'instancier une clé (`sac`) qui permet de chiffrer un octet en calculant un nombre entier et qui permet de déchiffrer un nombre entier en retrouvant l'octet d'origine.

Un octet est représenté par un tableau (type `list` en Python) dont les éléments sont les huit bits de l'octet.

Par exemple, le tableau `[1, 0, 1, 1, 1, 1, 0, 1]` représente l'octet `10111101`.

Cette classe est initialisée avec un unique attribut `sac` qui est un tableau renvoyé par la fonction `generer_sac` du module `outils`.

Afin de chiffrer un octet, on multiplie chaque bit d'indice `i` dans le tableau qui le représente par l'entier d'indice `i` du `sac` de l'instance de `Cle_symetrique` puis on renvoie la somme de tous les produits obtenus. Le chiffrement d'un octet est donc un entier.

Par exemple, si la clé est `[568, 205, 71, 32, 10, 6, 2, 1]`, on peut chiffrer `[1, 0, 1, 1, 1, 1, 0, 1]` par $568 \times 1 + 205 \times 0 + 71 \times 1 + 32 \times 1 + 10 \times 1 + 6 \times 1 + 2 \times 0 + 1 \times 1 = 688$.

13. Recopier et compléter la ligne 4 du code de la classe `Cle_symetrique` ci-après.

```
1 import outils
2 class Cle_symetrique():
3   def __init__(self):
4     ... = ...
```

14. Recopier et compléter les lignes 8 à 11 du code de la méthode `chiffrer` ci-après.

```
 1 def chiffrer(self, octet):
 2   '''
 3   renvoie le chiffrement de octet
 4   : paramètre octet (list) un tableau de huit 0 ou 1
 5   : return (int)
 6   '''
 7   tab_produits = []
 8   for i in range(...):
 9     produit = ... * ...
10     tab_produits.append(produit)
11   s = ...
12   return s
```

Afin de déchiffrer un entier et de retrouver l'octet auquel il correspond, on utilise la méthode `dechiffrer` ci-après.

Pour chaque indice `i` allant de 0 à 7 tous deux inclus :

- si l'entier à déchiffrer est supérieur ou égal à l'entier d'indice `i` du `sac` de l'instance de `Cle_symetrique` alors :
  - le bit d'indice `i` vaut 1 ;
  - on soustrait à l'entier à déchiffrer l'entier d'indice `i` du sac.
- sinon le bit d'indice `i` vaut 0.

15. Justifier qu'avec la méthode `dechiffrer`, le déchiffrement du nombre 688 est `[1, 0, 1, 1, 1, 1, 0, 1]`.

16. Écrire le code de la méthode `dechiffrer` de la classe `Cle_symetrique` qui prend en paramètre un `entier` à déchiffrer et renvoie un tableau représentant l'octet correspondant à cet `entier`.

    Exemple :

```
>>> ma_cle = Cle_symetrique()
>>> ma_cle.sac
[568, 205, 71, 32, 10, 6, 2, 1]
>>> ma_cle.dechiffrer(688)
[1, 0, 1, 1, 1, 1, 0, 1]
```