BACCALAURÉAT GÉNÉRAL

ÉPREUVE D'ENSEIGNEMENT DE SPÉCIALITÉ

SESSION 2025

NUMÉRIQUE ET SCIENCES INFORMATIQUES

JOUR 1

Durée de l'épreuve : **3 heures 30**

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

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

Ce sujet comporte 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 graphes, les protocoles réseaux et la programmation orientée objet.

Partie A

Le graphe suivant modélise un ensemble de routeurs ; les sommets sont les routeurs, les arêtes les liaisons entre ceux-ci.

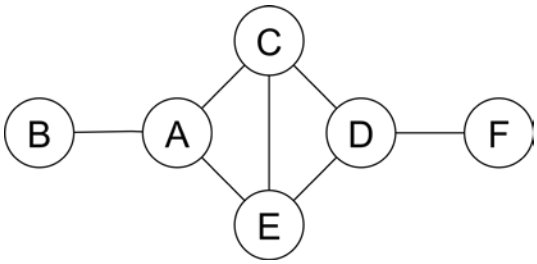

Figure 1. Schéma des routeurs et des liaisons

On désire parcourir ce graphe *en largeur* depuis le sommet A.

1. Dire lequel de ces parcours est un parcours en largeur en justifiant :
    - ABCDEF ;
    - ABCEDF ;
    - ABCDFE.

Voici un résumé sommaire du fonctionnement du protocole RIP permettant à chaque routeur d'un réseau de taille modérée d'établir sa table de routage :

Règle a (*règle d'initialisation*). Chaque routeur initialise sa table en y ajoutant ses voisins directs. Ils sont accessibles en un saut, sans passer par aucun routeur intermédiaire.

Règle b (*règle de transmission/réception*). À intervalles de temps réguliers chaque routeur envoie sa table de routage à ses voisins.

Règle c (*règle de mise à jour*). Lorsqu'un routeur reçoit les informations d'un routeur voisin, trois cas peuvent survenir :

- une route vers un nouveau routeur lui est présentée : il l'ajoute à sa table de routage ;
- une route vers un routeur déjà connu lui est présentée, plus longue en nombre de sauts que celle inscrite dans sa table : elle est ignorée ;
- une route vers un routeur déjà connu lui est présentée, mais strictement plus courte en nombre de sauts que la précédente : l'ancienne est remplacée par celle-ci.

La réception par chaque routeur des tables de tous ses voisins et la mise à jour de sa table de routage en conséquence constitue une *itération* du protocole. Au bout d'un petit nombre de ces itérations, plus aucune table de routage ne varie, on dit que le processus est *stabilisé*.

Pour tout cet exercice, on n'envisagera pas les cas problématiques dans lesquels une liaison est coupée ou un routeur tombe en panne.

On considère des routeurs A, B, C, D, E et F connectés comme indiqué sur le graphe de la figure 1.

Voici la table de routage de A à l'initialisation du protocole RIP :

Table de routage de A		
routeur	nombre de sauts	prochain routeur
B	1	–
C	1	–
E	1	–

2. Donner la table de routage de F à l'initialisation du protocole RIP.
3. Donner la table de routage de A après une première itération de RIP (deux réponses sont possibles).
4. Donner le numéro de l'itération de RIP à partir duquel les tables des routeurs du réseau ne varient plus.

On suppose dans la question suivante que les routeurs E et F sont reliés.

5. Donner la nouvelle table de routage de A après stabilisation de RIP (deux réponses sont possibles).

Partie B

Pour simuler la situation précédente et les tables de routage, on modélise le fonctionnement d'un routeur par la classe `Routeur`. Chaque instance `r` de la classe `Routeur` possède quatre attributs.

- `nom` : une chaine de caractères qui identifie le routeur.
- `voisins` : une liste d'objets de type `Routeur`. Il s'agit de routeurs qui sont directement connectés au routeur `r`.
- `nb_sauts` : un dictionnaire qui associe à chaque routeur accessible depuis `r` le nombre de sauts nécessaires pour l'atteindre depuis `r`.
- `prochain` : un dictionnaire qui associe à chaque routeur `r_accessible`, accessible depuis `r`, le premier routeur sur un chemin qui mène à

`r_accessible`, en n sauts, où n est la valeur associée à `r_accessible` dans `nb_sauts`. S'il y a un unique saut de `r` à `r_accessible`, alors la valeur associée au routeur `r_accessible` est `None`.

Initialement, tous les routeurs sont déconnectés : l'attribut `voisins` est initialisé avec la liste vide, et les attributs `nb_sauts` et `prochain` avec le dictionnaire vide.

```
class Routeur:
    def __init__(self, nom):
        self.nom = nom
        self.voisins = []
        self.nb_sauts = {}
        self.prochain = {}
```

Dans le programme principal, on crée les routeurs de la manière suivante :

```
A = Routeur('A')
B = Routeur('B')
C = Routeur('C')
D = Routeur('D')
E = Routeur('E')
F = Routeur('F')
```

Ainsi que la liste des routeurs

```
liste_routeurs = [A, B, C, D, E, F]
```

Afin de pouvoir relier les routeurs entre eux, on souhaite écrire une méthode `relie`, de la classe `Routeur`, dont on donne le code incomplet ci-dessous. Cette méthode prend en argument le routeur `self` ainsi qu'un routeur `autre` et met à jour si nécessaire les attributs `voisins`, `nb_sauts` et `prochain` des routeurs `self` et `autre` afin d'indiquer la présence d'une connexion entre ces deux routeurs. Dans le cas où les routeurs sont déjà connectés, cette méthode ne fait rien.

```
1     def relie(self, autre):
2         if autre not in self.voisins:
3             self.voisins.append(...)
4             self.nb_sauts[autre] = ...
5             self.prochain[autre] = ...
6             if self not in autre.voisins:
7                 autre.relie(...)
8
```

6. Recopier et compléter le code de la méthode `relie`.

7. Écrire la méthode `relie_liste` de la classe `Routeur` qui prend en paramètre une liste de routeurs `lst` et qui relie le routeur `self` à chacun des routeurs de la liste `lst`.

Par exemple, pour relier le routeur `A` aux routeurs `B`, `C` et `E`, on exécute l'instruction :

```
A.relie_liste([B, C, E])
# On n'appelle pas B.relie(A) car la liaison est déjà faite
```

8. Écrire les instructions manquantes pour relier les routeurs de manière à obtenir le graphe de la figure 1.

D'après la règle *c* (règle de mise à jour) du protocole RIP, lorsqu'un routeur reçoit les informations d'un routeur voisin, il doit mettre à jour sa table de routage. On donne ci-dessous le code incomplet de la méthode `met_a_jour_table` qui implémente la règle *c* du protocole RIP.

```
def met_a_jour_table(self, autre):
    for r in autre.nb_sauts:
        if r != self:
            if (r not in self.nb_sauts or
                self.nb_sauts[r] > ...):
                self.nb_sauts[r] = ...
                self.prochain[r] = ...
```

9. Recopier et compléter le code de la méthode `met_a_jour_table` ci-dessus.

10. Écrire la méthode `itere_rip` qui prend en paramètre le routeur `self` et met à jour sa table de routage lorsqu'il reçoit la table de routage de chacun des routeurs présents dans la liste de ses voisins.

11. Écrire une fonction qui prend en paramètre une liste de routeurs `l_routeurs` et qui réalise une itération du protocole RIP pour tous les routeurs de `l_routeurs`.

Au bout de quelques itérations, le protocole RIP converge : plus aucune table de routage du réseau n'est modifiée. On aimerait pouvoir itérer le protocole dans le programme principal jusqu'à ce que ce soit le cas, à l'aide d'une boucle `while`.

On suppose que la méthode `met_a_jour_table` de la classe `Routeur` a été modifiée de telle sorte qu'elle renvoie `True` dans le cas où le routeur `self` a procédé à une mise à jour de sa table de routage, et `False` sinon.

12. Écrire une version modifiée du code de la méthode `itere_rip` de la classe `Routeur` de telle sorte que celle-ci renvoie `True` dans le cas où le routeur `self` a procédé à une modification de sa table de routage au cours de l'exécution de la méthode `itere_rip`, et `False` sinon.

On donne ci-dessous le code du programme principal. On suppose que les instructions permettant de relier les routeurs ont été écrites à la suite et que la situation est celle décrite dans le graphe de la figure 1.

```
A = Routeur('A')
B = Routeur('B')
C = Routeur('C')
D = Routeur('D')
```

```
E = Routeur('E')
F = Routeur('F')
liste_routeurs = [A, B, C, D, E, F]
# instructions permettant de relier les routeurs
```

13. Compléter le code du programme principal afin que celui-ci mette à jour les tables de routage des routeurs présents dans la liste `liste_routeurs` jusqu'à ce qu'il ne soit plus nécessaire de faire des mises à jour des tables de routage.

    On ne demande pas de réécrire les instructions permettant de connecter les routeurs entre eux.

Exercice 2 (6 points)

Cet exercice porte sur les bases de données et la programmation orientée objet.

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` et `ORDER BY`.

Susie décide de créer une base de données qui recense des randonnées allant d'un parking à un lac.

Elle crée trois relations représentées sur le schéma ci-dessous (figure 1).

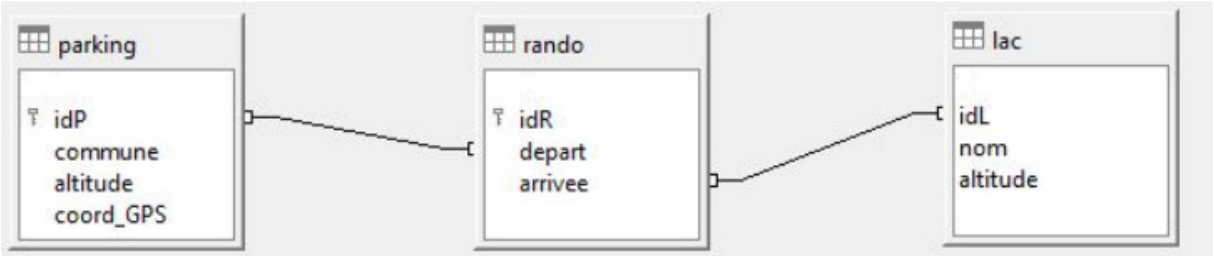

Figure 1. Schéma des trois relations

Les clés primaires sont signalées par une clé. Dans la relation `rando` :

- `depart` est une clé étrangère qui référence l'attribut `idP` de la relation `parking` ;
- `arrivee` est une clé étrangère qui référence l'attribut `idL` de la relation `lac`.

L'altitude, exprimée en mètre, est un entier.

Voici un extrait des enregistrements de ces trois relations.

parking			
idP	commune	altitude	coord_GPS
1	Chamonix	1 026	(45.98;6.89)
2	Argentiere	1 429	(45.99;6.92)
3	Passy	600	(45.92;6.72)
4	Passy	1 181	(45.95;6.71)
5	Nevache	2 022	(45.05;6.52)

rando		
idR	depart	arrivee
1	1	1
2	2	1
3	1	2
4	3	3

lac		
idL	nom	altitude
1	Lac Blanc	2 354
2	Lacs Noirs	2 564
3	Lac Vert	1 266
4	Lac Rouge	2 585

1. Indiquer ce que renvoie la requête suivante lorsqu'on l'applique aux extraits précédents.

```
SELECT nom
FROM lac
WHERE altitude <= 2000;
```

2. Indiquer les noms des lacs qu'on peut atteindre depuis le parking de Chamonix d'après la base de données de Susie.

À partir de maintenant, on travaille sur la totalité des enregistrements et non plus seulement sur les extraits précédents.

3. Donner une requête permettant d'obtenir les coordonnées GPS des parkings situés dans la commune de Passy à une altitude comprise strictement entre 800 et 1 000 mètres.

4. Donner une requête permettant d'obtenir les noms des lacs qu'il est possible d'atteindre depuis le parking situé à 1300 mètres d'altitude dans la commune de Cordon (on admet qu'un tel parking existe dans la base de données).

Dans les questions suivantes, l'ordre des requêtes SQL est important. On considère une nouvelle randonnée qui part du parking dont l'identifiant est 3 à Passy et qui conduit au lac d'Anterne situé à 2 059 mètres d'altitude.

Le parking est déjà dans la base de données mais par contre, ni la randonnée, ni le lac n'y figurent.

5. Donner les requêtes permettant à Susie d'ajouter à sa base de données cette randonnée et ce lac (on pourra utiliser l'identifiant 42 pour le lac et l'identifiant 100 pour la randonnée).

6. Susie a fait une erreur de saisie en insérant le nom du lac, elle a écrit 'Lc d Anterne'. Donner la requête permettant de corriger cette erreur.

7. Le parking dont l'identifiant est 28 a été transformé en un parc et n'existe plus.

   Donner les requêtes permettant de supprimer ce parking de la base de données.

Susie souhaite obtenir pour chacun des parkings le nombre de randonnées qui en partent.

Elle n'a pas encore appris à le faire en SQL et décide de le faire en Python. Pour cela elle définit la classe `Rando` ci-dessous permettant de représenter chacune des randonnées. La table `rando` est alors donnée par une liste d'objets de la classe `Rando`.

```
1 class Rando:
2     def __init__(self, idR, depart, arrivee):
3         self.idR = idR          # identifiant de la rando
4         self.depart = depart    # identifiant du parking
5         self.arrivee = arrivee  # identifiant du lac
```

8. Recopier et compléter les lignes 3 et 5 de la fonction `get_parking` qui prend en paramètre une liste de randonnées et qui renvoie la liste des identifiants des différents parkings, points de départ de ces randonnées (cette liste ne devra pas avoir de doublon).

   ```
   1 def get_parking(randos):
   2     parkings = []
   3     for ...:
   4         if rando.depart not in parkings:
   5             ...
   6     return parkings
   ```

   Par exemple, `get_parking([Rando(1, 1, 1), Rando(2, 2, 1), Rando(3, 1, 2)])` renvoie `[1, 2]`.

9. Recopier et compléter la ligne 4 de la fonction `get_nb_rando` qui prend en paramètres un identifiant de parking et une liste de randonnées, et qui renvoie le nombre de randonnées qui partent de ce parking.

   ```
   1 def get_nb_rando(parking, randos):
   2     nb = 0
   3     for rando in randos:
   4         if ...:
   ```

```
5             nb = nb + 1
6     return nb
```

Par exemple, `get_nb_rando(1, [Rando(1, 1, 1), Rando(2, 2, 1), Rando(3, 1, 2)])` renvoie 2.

10. Écrire une fonction `nb_rando_par_parking` qui prend en paramètre une liste de randonnées et qui renvoie un dictionnaire qui associe à chaque identifiant de parking le nombre de randonnées qui partent de ce parking.

    Par exemple, `nb_rando_par_parking([Rando(1, 1, 1), Rando(2, 2, 1), Rando(3, 1, 2)])` renvoie `{1:2, 2:1}`.

Exercice 3 (8 points)

Cet exercice porte sur l'algorithmique, la représentation binaire des entiers positifs et la programmation en langage Python.

Partie A : Modélisation du problème

On s'intéresse à un jeu de calcul mental appelé **Objectif somme**. Le jeu se joue sur un plateau de 5x5 cases. Chaque case contient un chiffre non nul de 1 à 9. On dispose également de nombres cibles en ligne et en colonne. Le but est de trouver les cases du tableau à vider afin d'atteindre les cibles en ligne et en colonne :

- sur chaque ligne, la somme des cases restantes doit valoir la cible de cette ligne ;
- sur chaque colonne, la somme des cases restantes doit valoir la cible de cette colonne.

De plus, il faut conserver au moins un chiffre par ligne.

Par exemple, la figure suivante représente un plateau de jeu et une solution :

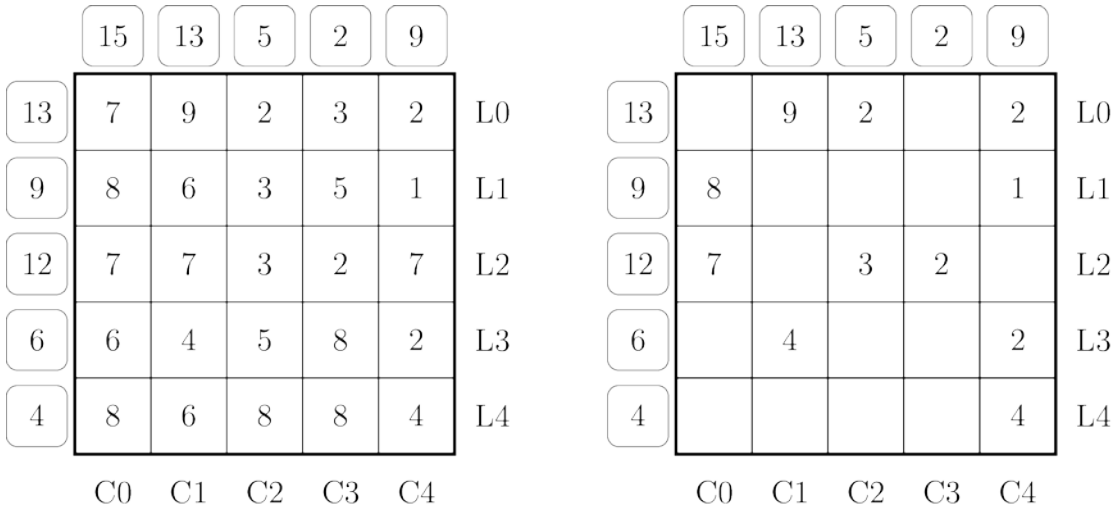

Figure 1. Plateau de jeu (à gauche) et une solution (à droite).

Les lignes du plateau seront nommées de *L0* à *L4* et les colonnes de *C0* à *C4*.

Ainsi l'exemple de la figure 1, la ligne *L1* fait référence aux valeurs 8,6,3,5,1 du plateau et la colonne *C3* fait référence aux valeurs 3,5,2,8,8 du plateau.

Dans la suite, on suppose que les cibles sont nécessairement des entiers entre 1 et 45.

1. Expliquer pourquoi on fait cette hypothèse.
2. Donner la plus petite valeur de cible que la ligne `[6,4,5,8,2]` peut atteindre. Donner aussi la plus grande valeur de cible que la ligne peut atteindre.

Dans la suite on appelle *plateau* une liste de 5 listes de 5 entiers. Chacune des listes de 5 entiers représente une ligne. Les entiers de ces listes sont compris entre 0 et 9, 0 représente une case vide. Pour représenter un jeu, un plateau doit être accompagné de deux listes de 5 entiers : la liste des cibles de lignes, la liste des cibles des colonnes.

Voici une représentation en langage Python de la figure 1 :

```
plateau_ex = [[7, 9, 2, 3, 2],
              [8, 6, 3, 5, 1],
              [7, 7, 3, 2, 7],
              [6, 4, 5, 8, 2],
              [8, 6, 8, 8, 4]]

ciblesLignes_ex = [13, 9, 12, 6, 4]

ciblesColonnes_ex = [15, 13, 5, 2, 9]
```

3. Écrire une fonction `extraireLigne` qui prend en paramètre un plateau et un indice *i* (*i* compris entre 0 et 4 inclus) et renvoie la ligne *Li* du plateau. Par exemple, la valeur de retour de l'appel `extraireLigne(plateau, 0)` est `[7, 9, 2, 3, 2]`.

4. Écrire une fonction `extraireColonne` qui prend en paramètre un plateau et un indice *i* (compris entre 0 et 4 inclus) et renvoie la colonne *Ci* du plateau. Par exemple, la valeur de retour de l'appel `extraireColonne(plateau, 1)` est `[9, 6, 7, 4, 6]`.

Partie B : Simplification du problème

Dans la figure 1, la solution comporte des cases vides. Ces cases correspondent aux chiffres que l'on a éliminés. En langage Python, on représentera ces cases vides par des zéros. Ainsi, pour éliminer du plateau un chiffre, il suffira de le remplacer par 0.

5. Donner la représentation en langage Python du plateau de la solution proposée.

On se propose d'utiliser deux règles pour éliminer simplement certains chiffres du plateau.

Règle 1 : on remarque que les chiffres d'une ligne ou d'une colonne donnée du plateau doivent être inférieurs ou égaux à la cible. Par exemple, pour la ligne *L4* de la figure 1, la cible est 4, on peut alors éliminer tous les chiffres 8 et 6. En appliquant la règle 1, *L4* devient alors `[0, 0, 0, 0, 4]`.

6. Pour le jeu représenté à gauche sur la figure 1, donner en Python le plateau obtenu en appliquant la règle 1 à chaque ligne.

La fonction à compléter `regle1` ci-dessous est une implémentation de la règle **1**. Elle prend en paramètre `plateau`, `ciblesLignes` et `ciblesColonnes` décrivant un jeu comme expliqué plus haut, et elle modifie `plateau` en appliquant la règle 1 à chaque ligne et à chaque colonne.

```
 1 def regle1(plateau, ciblesLignes, ciblesColonnes):
 2     for i in range(5):
 3         tab = extraireLigne(plateau, i)
 4         cible = ...
 5         for j in range(5):
 6             if tab[j] > cible:
 7                 plateau[i][j] = 0
 8     for j in range(5):
 9         tab = extraireColonne(plateau, j)
10         cible = ...
11         for i in range(5):
12             if tab[i] > cible:
13                 plateau[i][j] = 0
```

7. Recopier et compléter les lignes 4 et 10 pour compléter le code de la fonction `regle1`.

Règle 2 : S'il n'y a qu'un seul nombre impair dans une ligne ou une colonne dont la cible est paire, on peut éliminer ce nombre impair. Par exemple, pour la ligne *L3* de la figure 1, la cible est 6 et il n'y a qu'un nombre impair : 5. On peut donc éliminer ce 5.

8. Écrire une fonction `unImpair` qui prend comme paramètre une liste d'entiers, et qui renvoie `True` si la liste ne contient qu'un seul entier impair et `False` sinon.

La fonction à compléter `regle2` ci-dessous est une implémentation de la règle **2**. Elle prend en paramètre `plateau`, `ciblesLignes` et `ciblesColonnes` décrivant un jeu comme expliqué plus haut, et elle modifie `plateau` en appliquant la règle 2 à chaque ligne et à chaque colonne.

9. Recopier et compléter les lignes 4, 5, 11 et 12 pour compléter le code de la fonction **regle2** ci-dessous qui prend comme paramètre un `plateau`, une `ciblesLignes` et une `ciblesColonnes` et qui applique la règle **2**.

```
 1 def regle2(plateau, ciblesLignes, ciblesColonnes):
 2     for i in range(5):
 3         ligne = extraireLigne(plateau, i)
 4         ...
 5         if ...:
 6             for j in range(5):
 7                 if plateau[i][j] % 2 == 1:
 8                     plateau[i][j] = 0
 9     for j in range(5):
10         colonne = extraireColonne(plateau, j)
11         ...
12         if ...:
13             for i in range(5):
```

```
14                  if plateau[i][j] % 2 == 1:
15                      plateau[i][j] = 0
```

Ces règles permettent de simplifier le jeu mais pas de le résoudre dans tous les cas. Il est nécessaire d'utiliser d'autres méthodes.

Partie C : Problème sur une ligne et représentation binaire

Pour aider à la résolution du jeu **Objectif somme**, on cherche dans cette partie à résoudre le problème sur une ligne seulement. Il s'agit de trouver les nombres d'une liste de 5 entiers dont la somme est égale à un nombre cible.

Par exemple, une solution pour la liste `[6, 4, 5, 8, 2]` avec la cible 6 est de conserver le chiffre 6 uniquement. Une autre solution est de conserver le 4 et le 2.

On représente la première solution (conserver le 6) par la liste `[1, 0, 0, 0, 0]`. Cela signifie que la solution choisie est uniquement le premier élément de la liste. La liste `[1, 0, 0, 0, 0]` est appelée un **masque solution** du problème. Le masque solution correspondant à la solution avec le 4 et le 2, est alors `[0, 1, 0, 0, 1]`.

10. Expliquer pourquoi `[1, 1, 0, 0, 1]` est un masque solution pour la liste `[1, 2, 3, 5, 2]` et la cible 5. Donner tous les autres masques solutions.

11. Écrire une fonction `somme` qui prend comme paramètres une liste de 5 entiers et un masque (une liste de taille 5 de `0` et de `1`) et qui renvoie la somme des chiffres du tableau correspondant au masque. Par exemple, `somme([1, 5, 3, 4, 8], [0, 1, 1, 0, 1])` doit renvoyer 5 + 3 + 8 = 16.

On peut remarquer que les masques solutions correspondent à des nombres en écriture binaire. Par exemple, le masque `[0, 1, 0, 0, 1]` correspond à l'entier 9 car $0 \times 16 + 1 \times 8 + 0 \times 4 + 0 \times 2 + 1 \times 1 = 9$. Ainsi, on représente les masques possibles par des nombres en écriture binaire sur 5 bits.

12. Donner la représentation binaire sur 5 bits de l'entier 26 sous la forme d'une liste de taille 5.

13. Expliquer pourquoi on ne représente que les entiers compris entre 0 et 31 sur 5 bits.

14. Écrire une fonction `dec2bin` qui prend comme paramètre un entier compris entre 0 et 31 et qui renvoie sa représentation binaire sous la forme d'une liste de 5 bits. Par exemple la valeur de retour de l'appel `dec2bin(9)` est `[0, 1, 0, 0, 1]`.

Pour résoudre le problème, on se propose de générer tous les masques possibles avec la fonction `dec2bin` et de tester si ce sont des masques solutions. On stockera alors tous ces masques solutions dans une liste. On pourra utiliser la méthode `append` appliquée à une liste. Cette méthode permet d'ajouter un élément en fin de liste. Par exemple, à l'issue du code suivant, la liste `solutions` est `[1, 2]` :

```
solutions = [] # liste vide
solutions.append(1)
solutions.append(2)
```

15. Écrire une fonction `masques_solutions` qui prend comme paramètres une liste de taille 5 entiers et une cible, et qui renvoie la liste de tous les masques solutions correspondant.

Partie D : Retour au jeu “Objectif Somme”

Finalement, on vérifie qu’un plateau proposé comme solution respecte bien les contraintes sur les lignes et les colonnes.

16. Écrire une fonction `teste_solution` qui prend comme paramètres un plateau, la liste des cibles des lignes, la liste des cibles des colonnes, et qui retourne `True` si les valeurs des cases restantes du plateau vérifient bien les cibles (sur chaque ligne et sur chaque colonne), et `False` sinon.