BACCALAURÉAT GÉNÉRAL

ÉPREUVE D'ENSEIGNEMENT DE SPÉCIALITÉ

SESSION 2026

NUMÉRIQUE ET SCIENCES INFORMATIQUES

JOUR 2

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 18 pages numérotées de 1/18 à 18/18.

Le sujet est composé de trois exercices indépendants.

Le candidat traite les trois exercices.

Exercice 1 (6 points)

Cet exercice porte sur la programmation orientée objet et les structures linéaires.

Le *taquin* est un casse-tête qui se joue avec une grille de 9 cases : une case vide et 8 cases numérotées de 1 à 8.

Le jeu démarre en position gagnante représentée comme sur la figure 1.

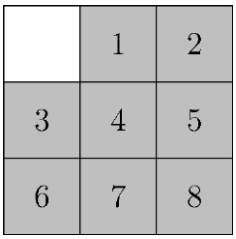


Figure 1. Position gagnante d'une grille

Les cases situées directement à gauche ou à droite ou en-dessous ou au-dessus de la case vide peuvent permuter avec elle. Afin de jouer au jeu taquin, on mélange la grille et l'objectif est de retrouver la position gagnante. On donne ci-dessous un exemple de grille mélangée.

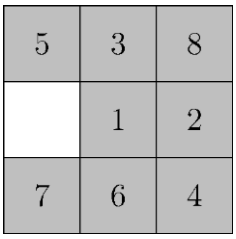


Figure 2. Exemple d'une grille mélangée

Par exemple, à partir de la situation de la figure 2, seules les cases numérotées 5, 1 ou 7 peuvent permuter leur position avec celle de la case vide.

On représente la grille d'un taquin à l'aide d'un tableau (type `list` en Python) dont les éléments sont les numéros des cases de la grille de gauche à droite et de haut en bas, la case vide portant le numéro 0. Le tableau ci-dessous contient les numéros de cases de la grille de la figure 2 :

```
tab = [5, 3, 8, 0, 1, 2, 7, 6, 4]
```

1. Donner l'indice de l'élément de `tab` correspondant à la case portant le numéro 2 de la grille.

   Dessiner la grille du taquin représentée par `tab` après l'exécution des instructions ci-dessous :

   ```
   1 tab[3] = tab[4]
   2 tab[4] = 0
   ```

3. Écrire une expression booléenne qui est évaluée à `True` si le taquin représenté par le tableau `tab` est en position gagnante et `False` sinon.

On souhaite utiliser une classe `Taquin` pour modéliser le jeu du taquin. On initialise cette classe de la façon suivante :

```
1 class Taquin :
2     def __init__(self):
3         self.tab = [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

4. Écrire la méthode `est_gagnant` de la classe `Taquin` qui renvoie `True` si l'instance de `Taquin` est en position gagnante et `False` sinon.

Voici le code partiel de la méthode `indice` qui renvoie l'indice de l'élément de l'attribut `tab` qui a pour valeur `numero` :

```
1     def indice(self, numero):
2         assert type(numero) == int, 'numero doit être entier'
3         ... , 'numero de case non valide'
4         i = 0
5         while ... :
6             i = i + 1
7         return i
```

5. Recopier et compléter les lignes 3 et 5 de la méthode `indice`.

On admet que la classe `Taquin` dispose d'une méthode `est_possible` qui prend en paramètre un entier `numero`, compris entre 1 et 8 inclus, et qui renvoie `True` s'il est possible de déplacer la case portant ce numéro et `False` sinon.

6. Recopier et compléter les lignes 2 à 6 de la méthode `jouer` ci-après qui, s'il est possible de déplacer le numéro passé en paramètre, intervertit la case portant ce numéro et la case vide.

```
1     def jouer(self, numero):
2         if ... :
3             i = self.indice(numero)
4             j = self.indice(0)
5             ... = numero
6             ... = ...
```

On admet que la classe `Taquin` possède une méthode `coups_possibles` qui renvoie un tableau dont les éléments sont les numéros des cases déplaçables. Si `t` est une instance de la classe `Taquin` qui correspond à celui de la figure 2, `t.coups_possibles()` renverra `[5, 1, 7]`.

On importe le module `random` et on rappelle que, si `tab` est du type `list`, l'instruction `random.choice(tab)` renvoie au hasard un élément de `tab`.

Pour mélanger n fois le taquin, on choisit au hasard successivement n fois un numéro parmi les coups possibles. On décide de ne pas jouer deux fois de suite le même numéro.

7. Recopier et compléter les lignes 4, 5, 6 et 9 de la méthode `melanger` donnée ci-dessous.

```
 1      def melanger(self, n):
 2          precedent = None
 3          i = 0
 4          while ... :
 5              possibilites = ...
 6              choix = ...
 7              if choix != precedent :
 8                  self.jouer(choix)
 9                  precedent = ...
10                  i = i + 1
```

On souhaite ajouter à l'implémentation du jeu de taquin un *mode résolution automatique* lors duquel le jeu jouera une série de numéros afin de revenir en position gagnante. On procède ainsi :

- au départ, la grille est en position gagnante, on initialise une pile vide et l'instance de la classe `Taquin` n'est pas en mode résolution automatique ;
- si l'instance n'est pas en mode résolution automatique, on empile les coups joués pour mélanger la grille ainsi que les coups joués par un joueur ;
- si l'instance est en mode résolution automatique, tant que le taquin n'est pas gagnant, on dépile la pile et on joue le coup associé.

Par exemple, à partir d'une grille en position gagnante, on considère que les numéros 1, 4, et 5 ont été joués dans cet ordre pour mélanger la grille. Ensuite le joueur a joué le numéro 2 puis a passé le jeu en mode résolution automatique.

8. Donner, pour la grille issue de l'exemple ci-dessus, les numéros des cases joués dans l'ordre lors de la résolution automatique en précisant à chaque étape l'état de la pile.

On dispose d'une classe `Pile` dont les méthodes sont `est_vide`, `empiler` et `depiler`. La méthode `est_vide` renvoie `True` si la pile est vide et `False` sinon. La méthode `empiler` prend en paramètre une valeur et l'empile dans la pile. La méthode `depiler` dépile une valeur de la pile si elle n'est pas vide et la renvoie.

On modifie l'initialisation de la classe `Taquin` de la façon suivante :

```
1 class Taquin :
2     def __init__(self):
3         self.tab = [0, 1, 2, 3, 4, 5, 6, 7, 8]
4         self.pile = Pile()
5         self.mode_resolution = False
```

On ajoute à la fin du code de la méthode `jouer` les deux lignes ci-dessous afin d'empiler les coups à chaque appel de cette méthode si l'instance du jeu n'est pas en mode résolution automatique :

```
7             if not self.mode_resolution :
8                 self.pile.empiler(numero)
```

Lorsque le joueur souhaite avoir la résolution automatique, il appelle la méthode `resoudre` qui :

- passe en mode résolution automatique en donnant la valeur `True` à l'attribut `mode_resolution` ;
- à chaque étape de la résolution automatique, affiche le coup joué.

9. Écrire le code de la méthode `resoudre`.

10. Expliquer pourquoi l'oubli du passage en mode résolution automatique dans la méthode `resoudre` entraînerait sa non terminaison.

Jouer deux fois de suite le même numéro est inutile. Pour éviter de stocker dans la pile deux tels coups, on souhaite modifier la méthode `jouer`.

Si l'instance du jeu taquin n'est pas en mode résolution automatique alors :

- si la pile du taquin n'est pas vide alors :
  - on récupère le numéro au sommet de la pile ;
  - si ce numéro et celui qu'on vient de jouer sont différents alors on les empile tous les deux dans le bon ordre, sinon on ne fait rien.
- sinon, on empile le numéro qu'on vient de jouer.

11. Proposer une modification de la méthode `jouer` à partir de la ligne 8 afin de prendre en compte cette optimisation.

Exercice 2 (6 points)

Cet exercice porte sur les bases de données et les graphes.

Partie A

Dans cette partie, on pourra utiliser les clauses du langage SQL pour :

- construire des requêtes d'interrogation à l'aide de `SELECT`, `FROM`, `WHERE` (avec les opérateurs logiques `AND` et `OR`), `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`.

On considère une table de données permettant d'enregistrer des informations concernant un jeu de société. Les joueurs s'inscrivent à l'aide d'un pseudonyme et peuvent participer à plusieurs parties. Plusieurs joueurs peuvent participer à une partie de ce jeu, ils peuvent alors gagner des points grâce à cette partie.

Cette base de données est composée des deux tables suivantes :

- `personne` :
  - `id_pers`, l'identifiant de la personne ;
  - `pseudo_pers`, le pseudonyme de la personne, c'est-à-dire un nom choisi par la personne qui s'inscrit ;
  - `date_pers`, la date d'inscription de la personne, au format text 'AAAA-MM-JJ' c'est-à-dire en spécifiant d'abord l'année puis le mois et enfin le jour.
- `participation` :
  - `id_partie`, l'identifiant d'une partie d'un jeu ;
  - `id_pers`, l'identifiant d'une des personnes qui a joué cette partie ;
  - `nb_point`, le nombre de points gagnés par cette personne à cette partie.

1. Donner la table dans laquelle l'attribut `id_pers` est une clé étrangère.
2. Justifier que la clé primaire de la table `participation` ne peut pas être `id_partie`.
3. Donner une requête permettant d'ajouter une personne dont le pseudonyme est `theorie` et dont la date d'inscription est le 14 décembre 2022. On prendra l'identifiant 42 pour cette personne.

4. Donner une requête permettant d'obtenir tous les identifiants des parties jouées par la personne dont le pseudonyme est `test` (on admettra qu'une seule personne possède ce pseudonyme).

5. Donner les requêtes permettant de supprimer la personne dont l'identifiant est 8 de la base de données.

Partie B

On s'intéresse au jeu suivant.

On considère un alphabet composé uniquement des voyelles `a`, `e`, `i`, `o`, `u`, `y`. Dans toute la suite, les mots considérés seront composés uniquement de ces 6 voyelles. Le jeu mélange en secret ces 6 voyelles pour obtenir un nouvel ordre alphabétique comme `ouyeai`. Ensuite, le jeu propose au joueur une liste de couples de mots distincts tels que le premier mot du couple est classé, suivant le nouvel ordre alphabétique, avant le deuxième mot. L'objectif du jeu est de proposer un ordre sur les 6 voyelles qui soit compatible avec ce classement.

On rappelle que, dans l'ordre lexicographique, la comparaison de deux mots s'effectue lettre par lettre, en commençant par la première, puis la suivante, etc. jusqu'à obtenir une différence. C'est la première lettre qui diffère qui donne l'ordre des deux mots.

Par exemple, si l'ordre alphabétique des lettres est `ouyeai`, alors

- le mot `ouu` est classé avant le mot `aei` car `o` est avant `a` dans `ouyeai` ;
- le mot `ouy` est classé avant le mot `oue` car les deux premières lettres sont identiques et `y` est avant `e` ;
- le mot `ou` est classé avant le mot `oue` car `ou` est un préfixe du mot `oue`.

6. Recopier et compléter les lignes 3 et 4 du code de la fonction `indice` qui prend en paramètres deux chaînes de caractères : `lettre` qui est une des 6 voyelles et `ordre` qui est un mélange des 6 voyelles, et qui renvoie l'indice où se trouve `lettre` dans `ordre`.

```
1 def indice(lettre, ordre):
2     for i in range(len(ordre)):
3         if ...:
4             return ...
```

7. Recopier et compléter les lignes 7, 9 et 11 du code de la fonction `comparer` qui prend en paramètres deux chaînes de caractères `mot1` et `mot2` représentant deux mots distincts ainsi qu'une chaîne de caractères `ordre` représentant un ordre alphabétique composé des 6 voyelles. La fonction renvoie `True` si `mot1` est classé avant `mot2` selon l'ordre `ordre` et `False` sinon.

```
 1 def comparer(mot1, mot2, ordre):
 2     i = 0
 3     while i < len(mot1) and i < len(mot2):
 4         i1 = indice(mot1[i], ordre)
 5         i2 = indice(mot2[i], ordre)
 6         if i1 < i2:
 7             return ...
 8         elif i1 > i2:
 9             return ...
10         i += 1
11     return ...
```

On s'intéresse maintenant à une stratégie pour atteindre l'objectif du jeu. Lorsque le joueur récupère la liste des couples de mots, il en déduit que certaines lettres sont placées avant d'autres. Par exemple, avec le couple de mots (`oou`, `ooai`), il peut déduire que la lettre `u` est placée avant la lettre `a`.

Ensuite, il construit un graphe orienté de la manière suivante : les sommets sont les 6 voyelles et à chaque fois qu'il déduit qu'une lettre (par exemple `e`) est avant une autre lettre (par exemple `a`), il va ajouter un arc partant de `e` vers `a` (s'il n'existait pas déjà).

Par exemple, on suppose que le jeu fournit la liste `mots_exemple` :

```
mots_exemple = [("oou", "ooai"), ("yee", "ieo"),
("ye", "aaaa"), ("a", "ieo"), ("euee", "eyy"),
("ao", "au"), ("e", "a")]
```

Le joueur en déduit alors que `u` est avant `a`, `y` est avant `i`, `y` est avant `a`, `a` est avant `i`, `u` est avant `y`, `o` est avant `u` et que `e` est avant `a`. Il construit alors le graphe orienté de la figure 1.

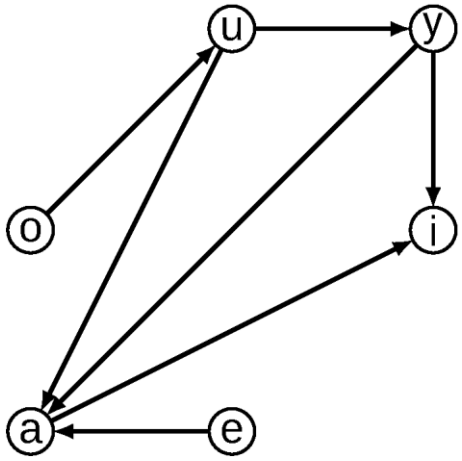


Figure 1. Graphe de l'exemple

8. Écrire le code de la fonction `premiere_diff` qui prend en paramètres deux chaînes de caractères `mot1` et `mot2` représentant deux mots distincts et qui renvoie le premier indice où les deux lettres sont différentes. Si l'un des deux mots est un préfixe de l'autre, la fonction renvoie la taille du préfixe.

   Exemples :

```
>>> premiere_diff("oou", "ooai")
2
>>> premiere_diff("aio", "aioiee")
3
```

La fonction `dico_adj`, dont le code est donné ci-dessous, prend en paramètre une liste `mots` représentant une liste des couples de mots distincts donnés par le jeu et renvoie un dictionnaire `adj` représentant les listes d'adjacence du graphe.

```
def dico_adj(mots):
    adj = {}
    for (mot1, mot2) in mots:
        ident = premiere_diff(mot1, mot2)
        if ident < len(mot1) and ident < len(mot2):
            petite = mot1[ident]
            grande = mot2[ident]
            if petite not in adj:
                adj[petite] = [grande]
            else:
                adj[petite].append(grande)
    return adj
```

9. Donner le dictionnaire obtenu après l'appel `dico_adj(mots_exemple)` où `mots_exemple` est la liste définie précédemment dans l'énoncé.

La fonction `parcours` réalise un parcours du graphe. Ses paramètres d'entrée sont :

- un dictionnaire `adj` représentant les listes d'adjacence d'un graphe ;
- une chaîne de caractères `s` représentant un sommet du graphe ;
- une liste `deja_vus` contenant les sommets du graphe qui ont déjà été ajoutés aux sommets à traiter ;
- une liste `tri` qui sera vide lors du premier appel puis qui va s'agrandir d'une lettre à chaque appel récursif.

```
def parcours(adj, s, deja_vus, tri):
    if s in adj:
        for v in adj[s]:
            if v not in deja_vus:
                deja_vus.append(v)
                parcours(adj, v, deja_vus, tri)
    tri.append(s)
```

10. Indiquer de quel type de parcours il s'agit.

11. Écrire une fonction `trier` qui prend en paramètre une liste `mots` représentant la liste des couples de mots et qui renvoie un ordre des 6 voyelles compatible avec le classement des mots donnés dans la liste.

    Pour cela, on suivra l'algorithme suivant :

    – récupérer le dictionnaire des listes d'adjacence du graphe ;
    – initialiser une liste vide appelée `tri` ;
    – initialiser une liste vide appelée `deja_vus` ;
    – pour chaque voyelle qui n'est pas dans `deja_vus`, l'ajouter à `deja_vus` et lancer le parcours depuis ce sommet ;
    – lorsque toutes les lettres ont été parcourues, inverser les éléments de la liste `tri` et la renvoyer.

    On pourra utiliser `liste.reverse()`. La méthode `reverse` ne renvoie rien mais modifie `liste` en inversant ses éléments.

Exercice 3 (8 points)

Cet exercice porte sur la programmation Python et le routage RIP.

On étudie dans cet exercice des robots qui peuvent se déplacer et communiquer entre eux.

Partie A : Gestion des déplacements du robot

Chaque robot peut se déplacer en avant et tourner de 90 degrés à droite ou à gauche. Il reçoit ses instructions sous la forme d'une chaîne de caractères contenant des lettres et éventuellement des entiers et des parenthèses. S'il lit un 'A', le robot avance d'une distance correspondant à "un pas" (valeur fixée dans les réglages du robot). Il tourne à droite s'il lit un 'D' et à gauche s'il lit un 'G'.

Insérer un entier dans la chaîne permet de répéter une action ou une suite d'actions écrites entre parenthèses.

'12A' est équivalent à 'AAAAAAAAAAAA', '3(AD)' est équivalent à 'ADADAD'.

La figure 1 représente le parcours du robot lorsqu'il reçoit la chaîne '3ADA'

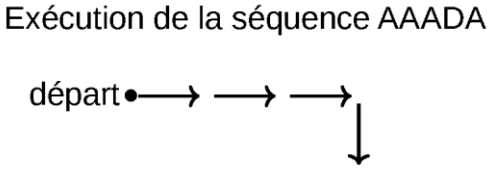


Figure 1. Exemple de parcours

1. Représenter le parcours effectué par le robot s'il reçoit la chaîne '4(AG)'.

Les premières fonctions exécutées vérifient que la chaîne transmise au robot ne comporte pas d'erreurs, en particulier que :

- tous les caractères de la chaîne sont des caractères inclus dans '0123456789ADG()' ;
- on n'a pas d'entier en fin de chaîne, ni d'entier juste avant une parenthèse fermante ;
- le parenthésage est correct (chaque parenthèse ouverte doit être refermée, dans le bon ordre).

Dans toute cette partie, l'opérateur `in` teste la présence ou l'absence d'un élément dans une chaîne de caractères. On peut également le combiner avec l'opérateur `not`.

Exemples :

```
>>> 'a' in 'bateau'
True
```

```
>>> 'y' in 'bateau'
False
>>> 'a' not in 'bateau'
False
>>> 'y' not in 'bateau'
True
```

2. Recopier et compléter la ligne 4 du code de la fonction `caracteres_valides` qui prend en paramètre une chaîne de caractères et renvoie un booléen indiquant si tous les caractères de la chaîne sont valides.

```
1 def caracteres_valides(chaine):
2     valides = '0123456789ADG()'
3     intrus = [c for c in chaine if c not in valides]
4     ... # Vrai si la liste est vide, faux sinon
```

3. Recopier et compléter les lignes 4, 8 et 9 du code de la fonction `entiers_valides` qui prend en paramètre une chaîne de caractères et renvoie un booléen indiquant si tous les entiers sont bien placés dans la chaîne.

```
1 def entiers_valides(chaine):
2     chiffres = '0123456789'
3     if chaine[len(chaine)-1] in chiffres:
4         ...
5     for indice in range(1,len(chaine)):
6         if chaine[indice] == ')':
7             if chaine[indice - 1] in chiffres:
8                 ...
9     ...
```

Pour déterminer si le parenthésage est correct, on propose l'algorithme suivant :

- on initialise une variable `parenthese` à la valeur 0 ;
- on parcourt les caractères de `chaine`, et pour chaque caractère lu,
  - si le caractère lu est `'('`, on augmente `parenthese` de 1 ;
  - si le caractère lu est `')'`, on diminue `parenthese` de 1 ;
  - si `parenthese` est strictement négatif alors la fonction renvoie `False` ;
- si le parcours s'achève, la fonction renvoie `True` si `parenthese` vaut 0 et `False` sinon.

4. Écrire le code de la fonction `parenthesage_correct` qui prend en paramètre une chaîne de caractères et qui renvoie un booléen indiquant si le parenthésage de la chaîne de caractères est correct en utilisant l'algorithme décrit ci-dessous.

La fonction `lire_nombre` prend en paramètres une chaîne de caractères `chaine` et un indice `indice` et renvoie la valeur du nombre dont l'écriture commence à `indice` ainsi que l'indice de son dernier chiffre.

```
1 def lire_nombre(chaine, indice):
2     chiffres = "0123456789"
3     nombre = ''
4     while chaine[indice] in chiffres:
5         nombre = nombre + chaine[indice]
6         indice = indice + 1
7     return (int(nombre), indice-1)
```

Exemple :

```
>>> lire_nombre('AD179AGA', 2)
(179, 4)
```

5. Lors de cet appel, la boucle `while` réalise trois itérations. Donner les valeurs des variables `nombre` et `indice` à la fin de chacune de ces itérations.

La fonction `lire_bloc` prend en paramètres une chaîne de caractères et l'indice d'une parenthèse ouvrante. Elle renvoie le contenu du bloc délimité par cette parenthèse et la parenthèse fermante associée ainsi que l'indice de cette parenthèse fermante.

Exemples :

```
>>> lire_bloc('2(AD)A', 1)
('AD', 4)
>>> lire_bloc('2(AD)3(2AGA)', 6)
('2AGA', 11)
>>> lire_bloc('2(3(AD)G)2A', 1)
('3(AD)G', 8)
```

6. Recopier et compléter les lignes 7, 8 et 9 de la fonction `lire_bloc` donnée ci-dessous.

```
 1 def lire_bloc(chaine, indice):
 2     indice = indice + 1
 3     caractere = chaine[indice]
 4     bloc = ""
 5     compteur = 1
 6     while compteur > 0:
 7         bloc = ...
 8         indice = ...
 9         caractere = ...
10         if caractere == '(':
11             compteur = compteur + 1
12         if caractere == ')':
13             compteur = compteur - 1
14     return (bloc, indice)
```

On suppose que la fonction `execute_mouvement` est déjà implémentée. Cette fonction prend en paramètre un caractère (`A`, `D` ou `G`) et exécute l'action du robot correspondante.

La fonction récursive `lire_parcours` permet de déplacer le robot suivant la chaîne de caractères donnée en paramètre. On donne ci-dessous le code incomplet de cette fonction.

```
 1 def lire_parcours(chaine):
 2     chiffres = "0123456789"
 3     indice = 0
 4     nombre = 1
 5     while indice < len(chaine):
 6         car_lu = chaine[indice]
 7         if car_lu in 'AGD':       # commande simple
 8             for k in range(nombre):
 9                 execute_mouvement(car_lu)
10             nombre = 1
11         elif car_lu in chiffres:  # répétition
12             t = ...
13             nombre = t[0]
14             indice = t[1]
15         elif car_lu == '(':       # début d'un bloc
16             t = ...
17             bloc = t[0]
18             indice = t[1]
19             for k in range(nombre):
20                 ...
21             nombre = 1
22         indice = indice + 1
```

7. Recopier et compléter les lignes 12, 16 et 20 du code de la fonction `lire_parcours`.

Partie B : Communication et routage

On considère un réseau de robots. Chaque robot peut communiquer par radio avec les robots qui se trouvent à proximité.

Si deux robots sont assez proches l'un de l'autre, ils peuvent communiquer directement. Lorsque deux robots sont trop éloignés l'un de l'autre, le message est relayé de robot en robot jusqu'à son destinataire.

On représente un réseau de robots par un graphe dont les arêtes symbolisent la possibilité de communiquer entre deux robots.

Exemple :

Dans la configuration ci-dessous, si le robot numéro 4 veut envoyer un message au robot 12, il devra le transmettre au robot 46 qui le transmettra alors au robot 12.

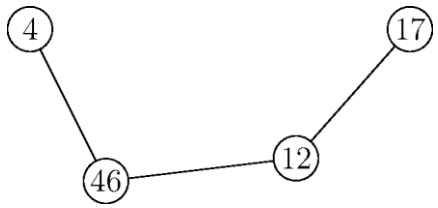


Figure 2. Un réseau constitué de quatre robots

Le routage des messages est effectué à l'aide du protocole RIP qui minimise la distance calculée en nombre de sauts réalisés entre deux routeurs. Chaque robot joue le rôle d'un routeur : il maintient une table de routage dans laquelle figurent les informations permettant de joindre les autres robots connus.

Une table de routage contient trois colonnes :

- la colonne "destination" donne l'identifiant du robot à atteindre ;
- la colonne "prochain robot" donne l'identifiant du prochain robot sur le chemin vers la destination ;
- la colonne "distance" indique le nombre total de transmissions à effectuer pour joindre la destination.

Exemple :

Le réseau suivant est constitué par les robots 46, 4, 57, 91, 22 disposés comme dans le graphe ci-dessous.

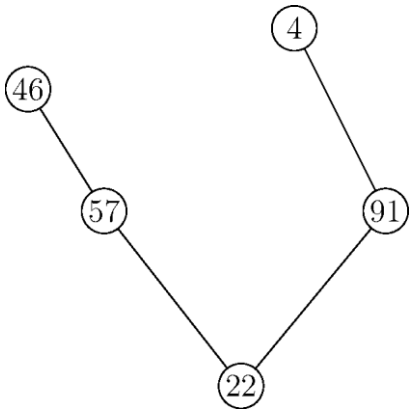


Figure 3. Réseau constitué par les cinq robots

La table de routage du robot 91 est :

destination	prochain robot	distance
4	4	1
46	22	3
22	22	1
57	22	2

On suppose maintenant qu'à la suite d'un déplacement du robot 4, une liaison directe avec le robot 46 est possible :

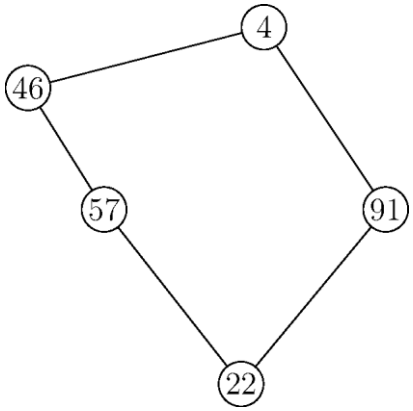


Figure 4. Le même réseau après mouvement du robot 4

8. Indiquer les modifications apportées à la table de routage du robot 91.

Un nouveau robot, portant l'identifiant 87, rejoint le réseau en contactant le robot numéro 91. Il lui envoie sa table de routage :

Table de routage du robot 87 :

destination	prochain robot	distance
91	91	1
63	63	1
36	63	2

9. Décrire les nouvelles modifications à apporter à la table de routage du robot 91.

Partie C : Programmation du routage

Afin de gérer la communication entre les robots du réseau, on implémente la classe `Module_comm` et chaque robot possède une instance de cette classe.

```
1 class Module_comm:
2     def __init__(self, id_robot):
3         self.identifiant = id_robot
4         self.table_routage = {}
```

Chaque clé du dictionnaire `table_routage` est l'identifiant d'un robot du réseau. La valeur associée est un dictionnaire donnant le premier robot sur le chemin vers ce robot ainsi que le nombre total de transmissions c'est-à-dire le nombre de sauts à effectuer.

La table du routage du robot 91 de la figure 3 est implémentée comme suit :

```
1 {
2     4  : { "prochain": 4, "distance": 1},
3     46 : { "prochain": 22, "distance": 3},
4     22 : { "prochain": 22, "distance": 1},
5     57 : { "prochain": 22, "distance": 2},
6 }
```

La méthode `ajouter_voisin` prend en paramètre l'identifiant d'un robot avec lequel il peut directement communiquer puis crée une entrée dans le dictionnaire `table_routage`.

10. Écrire le code de la méthode `ajouter_voisin`.

La méthode `nombre_sauts` prend en paramètre l'identifiant d'un robot et renvoie le nombre de sauts nécessaires pour communiquer avec ce robot. Pour cette implémentation, on fixe le nombre de sauts maximum à 15.

Si l'identifiant reçu par `nombre_sauts` ne figure pas dans la table de routage, la fonction renvoie la valeur 16.

On rappelle que les opérateurs `in` et `not in` permettent de savoir si un objet figure dans les clés d'un dictionnaire Python.

Exemple :

```
>>> D = {4  : { "prochain": 31, "distance": 2},
         46 : { "prochain": 22, "distance": 3} }
>>> 4 in D
True
>>> 5 not in D
True
```

11. Recopier et compléter la ligne 5 du code de la méthode `nombre_sauts` :

```
1     def nombre_sauts(self, identifiant):
2         if identifiant not in self.table_routage:
3             return 16
4         else:
5             return ...
```

12. Écrire le code de la méthode `voisins` qui renvoie la liste de tous les robots avec lesquels il peut directement communiquer.

Afin de ne pas communiquer d'informations inutiles, on souhaite communiquer à un robot voisin l'extrait de la table de routage qui contient uniquement les lignes pour lesquelles ce robot voisin n'est lui-même pas le prochain robot sur la route.

La méthode `communiquer_extrait_table` prend en paramètre l'identifiant d'un robot `voisin` et communique cet extrait de la table de routage. L'extrait de table renvoyé est de type dictionnaire et il possède la même structure que la table de routage complète.

13. Écrire le code de la méthode `communiquer_extrait_table`.