# Centres étrangers – 2025 – sujet2

## Exercice 3 (8 points)

*Cet exercice porte sur les graphes, la programmation objet et la récursivité.*

On considère un jeu où les nombres entiers de 1 à 9 sont placés autour d'un cercle. Le but du jeu est de relier le plus de nombres possibles en partant de l'un d'entre eux et en appliquant la règle suivante : le nombre suivant est un multiple ou un diviseur du précédent, sans jamais utiliser deux fois le même nombre. Par exemple, en partant du nombre 4 nous pouvons aller :

- soit au nombre 8 (qui est un multiple de 4) ;
- soit au nombre 2 (qui est un diviseur de 4) ;
- soit au nombre 1 (qui est un diviseur de 4).

Bien que 4 soit à la fois multiple et diviseur de 4, nous ne pouvons pas relier 4 à 4. C'est pourquoi dans la suite nous appellerons **diviseur** (respectivement **multiple**) d'un entier ses diviseurs (respectivement ses multiples) différents de lui-même. On dira donc que les seuls diviseurs de 4 sont 1 et 2, et que son seul multiple (entre 1 et 9) est 8.

Depuis le nombre 4, on peut donc choisir d'aller en 8, puis en 2 qui est diviseur de 8, puis 6 qui est multiple de 2. On construit ainsi le chemin 4, 8, 2, 6 de longueur 3. Mais puisque le but du jeu est de trouver un chemin le plus long possible, on a tout intérêt à prolonger ce chemin : on peut encore aller en 1 qui est diviseur de 6, puis en 7 qui est multiple de 1. On obtient ainsi le chemin 4, 8, 2, 6, 1, 7 représenté sur la figure 1.

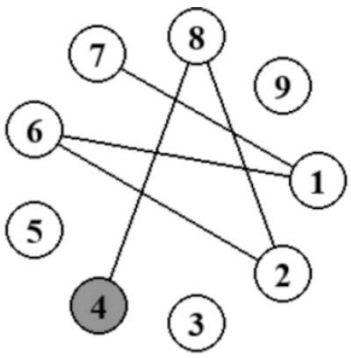


Figure 1. Un chemin de longueur 5 depuis le nombre 4

Ce chemin n'est pas prolongeable : en effet, ce chemin finit par le nombre 7, or 7 n'a aucun multiple entre 1 et 9, et un seul diviseur, 1, qui figure déjà dans le chemin.

Dans la suite nous appelons donc **chemin non prolongeable** une séquence d'entiers entre 1 et 9 :

- dans laquelle chaque entier (hors du premier) est multiple ou diviseur du précédent ;
- qui ne fait pas apparaître deux fois le même entier ;
- qui ne peut pas être prolongée en respectant les deux premiers points.

Ainsi le but du jeu est de trouver un chemin non prolongeable le plus long possible, en partant de n'importe quel nombre.

Les figures 2 et 3 donnent d'autres exemples de chemins non prolongeables possibles dans ce jeu.

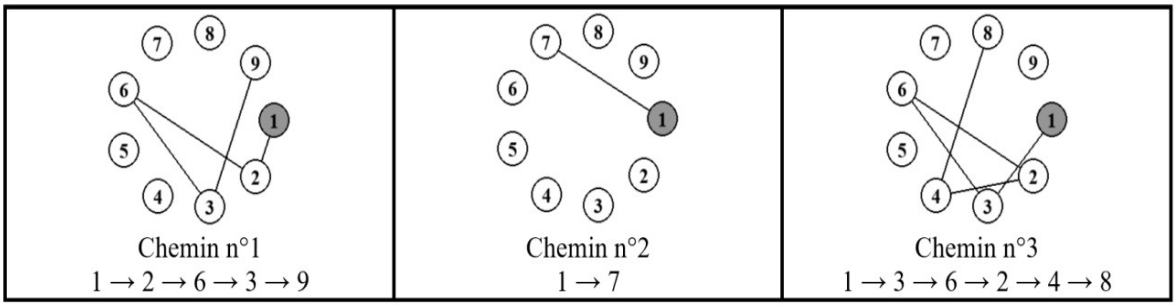


Figure 2. Trois chemins non prolongeables possibles en partant du nombre 1

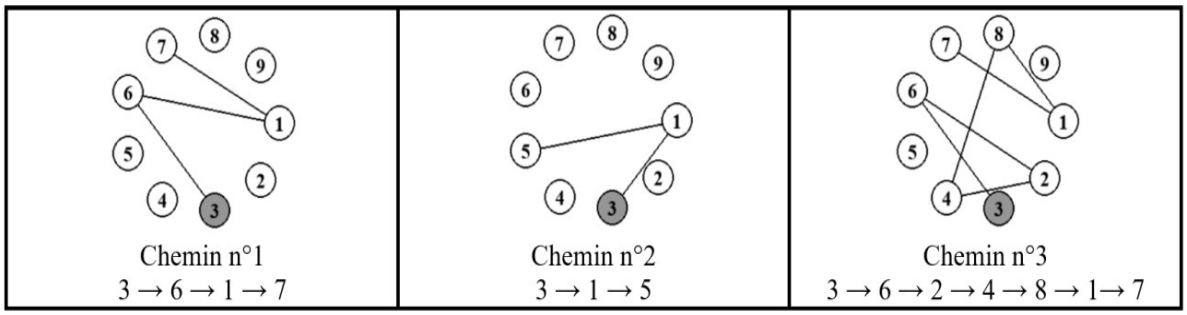


Figure 3. Trois chemins non prolongeables possibles en partant du nombre 3.

1. Donner un chemin non prolongeable qui commence par $3 \to 9 \to 1 \to 2$.

## PARTIE A : MODÉLISATION DU JEU PAR UN GRAPHE

Afin de représenter le jeu, nous allons construire le graphe à 9 sommets représentant les nombres entiers de 1 à 9, où chacun d'eux est relié à tous ses diviseurs et à tous ses multiples. Ce graphe non orienté est représenté en figure 4.

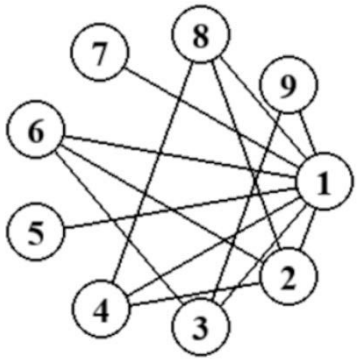


Figure 4. Le graphe du jeu à 9 entiers Nous allons implémenter ce graphe en Python en définissant tout d'abord une classe `Sommet` dont la méthode d'initialisation est donnée ci-dessous.

```
class Sommet :
    def __init__(self, valeur) :
        """Crée un sommet qui n'est relié à aucun autre"""
        self.valeur = valeur
        self.diviseurs = []
        self.multiples = []
```

L'attribut `valeur` donne l'entier représenté par ce sommet. L'attribut `diviseurs` (respectivement `multiples`) du sommet représentant l'entier $nn$ est une liste qui contiendra tous les objets de type `Sommet` représentant un diviseur (respectivement un multiple) de l'entier $nn$.

Nous pouvons alors représenter le graphe du jeu à 9 entiers par une liste de neuf sommets (neuf instances de la classe `Sommet`) représentant les entiers de 1 à 9, avec leurs diviseurs et leurs multiples entre 1 et 9. On appellera cette liste `jeu_9`.

Afin de créer un tel graphe de jeu, on utilise la fonction `creer_jeu` qui prend en paramètre un nombre entier positif `n` représentant le nombre de sommets du graphe et qui renvoie une liste de `n` instances de la classe `Sommet`. Considérons le code cidessous qui donne une première version de la fonction `creer_jeu` (ce code sera complété par la suite).

```
1     def creer_jeu(n):
2     """Renvoie le graphe du jeu à n sommets"""
3     jeu = []
4     for valeur in range(1, n+1):
5         sommet = Sommet(valeur)
6         jeu.append(sommet)
7     return jeu
8
9 jeu_9 = creer_jeu(9)
```

2. Donner les valeurs prises par la variable locale `valeur` lors de l'exécution de `creer_jeu(9)`.

3. Une fois ce code exécuté, donner la valeur de `jeu_9[0].valeur`.

Cette version de la fonction `creer_jeu` crée un graphe de jeu avec un sommet pour chaque entier, mais où chacun de ces sommets est isolé car aucun ne connaît ses diviseurs ni ses multiples (aucune arête n'a été créée). Afin de compléter ce graphe, il convient de relier chaque sommet à tous ses diviseurs, c'est pourquoi nous ajoutons à la classe `Sommet` la méthode `relier_diviseurs`, dont le code est donné cidessous

```
1      def relier_diviseurs(self, jeu):
2          """Relie ce sommet à tous les sommets de jeu
3          qui représentent des diviseurs de ce sommet"""
4          for s in jeu:
5              if s != self and self.valeur % s.valeur == 0:
6                  self.diviseurs.append(s)
7                  s.multiples.append(self)
```

On rappelle qu'en Python l'expression `a % b` renvoie le reste de la division entière de `a` par `b`. Par exemple :

```
>>> 8 % 4
0
>>> 8 % 5
3
```

4. Expliquer le test réalisé à la ligne 5 de la fonction `relier_diviseurs`.

5. Donner le contenu de `jeu_9[5].diviseurs` après l'exécution de `jeu_9[5].relier_diviseurs(jeu)`.

6. Écrire la ligne 6 manquante dans le code ci-dessous de la nouvelle version de la fonction `creer_jeu`.

```
1 def creer_jeu(n):
2      """ renvoie le graphe du jeu à n sommets"""
3      jeu = []
4      for valeur in range(1, n+1) :
5          sommet = Sommet(valeur)
6          ...
7          jeu.append(sommet)
8      return jeu
```

Pour vérifier la bonne implémentation du jeu, nous ajoutons à la classe `Sommet` deux autres méthodes : `lister_diviseurs` et `lister_multiples`. Ces méthodes renverront respectivement la liste des valeurs des diviseurs et des multiples pour un sommet donné sous la forme d'une liste de nombres entiers.

Voici des exemples d'utilisation de ces méthodes (toujours avec notre jeu à 9 sommets) :

```
>>> jeu_9[7].lister_diviseurs()
[1, 2, 4]
>>> jeu_9[1].lister_multiples()
[4, 8]
```

7. Recopier et compléter la ligne 3 du code de la méthode `lister_diviseurs` donné ci-dessous.

```
1           def lister_diviseurs(self):
2               """Renvoie la liste des diviseurs de ce sommet"""
3               return [... for sommet in ...]
```

On suppose que la méthode `lister_multiples` est codée de même.

8. Recopier et compléter le jeu de tests ci-dessous qui a pour but de vérifier que le sommet représentant le nombre 3 est bien relié à tous ses voisins dans le graphe `jeu`.

```
l_div_3 = ...
l_mult_3 = ...
assert 1 in l_div_3    assert ... in ...
assert ... in ...
```

On rappelle que l'instruction **`assert`** `condition` produit une erreur lorsque que `condition` est une expression booléenne qui vaut `False`.

## PARTIE B : STRUCTURE DE FILE

Nous verrons en partie C comment écrire le code qui permet de résoudre notre jeu. Nous aurons besoin pour cela d'une structure de données de type *file* (premier entré, premier sorti). La classe `File` présentée ci-après implémente cette structure de données à l'aide d'une liste Python. Les éléments ajoutés à la file sont ajoutés en dernière position de la liste. Les éléments supprimés de la file ne sont pas supprimés de la liste (pour des raisons d'efficacité), mais seulement ignorés. En effet grâce à l'attribut `decalage`, on considère que la tête de file est dans la liste à l'indice `decalage` et les éléments d'indices plus petits sont ignorés. Ainsi lorsqu'on veut supprimer la tête de la file, il suffit de décaler d'un cran le début de la file, c'est-à-dire d'ajouter un à l'attribut `decalage`.

```
class File :
    def __init__(self):
        """Crée une file vide"""
        self.donnees = []
        self.decalage = 0
    def est_vide(self):
        """Renvoie un booléen indiquant si la file est vide."""
        return self.decalage == len(self.donnees)
    def enfiler(self,element):
        """Insère element dans la file"""
        self.donnees.append(element)
    def defiler(self):
    """Retire un élément de la file et le renvoie"""
        assert not self.est_vide()
        tete = self.donnees[self.decalage]
        self.decalage = self.decalage + 1
        return tete
    def taille(self):
        """Renvoie le nombre d'éléments présents dans
        la file"""
        ...
```

9. Dans la méthode `defiler` expliquer la ligne de code suivante :

```
assert not self.est_vide()
```

10. Recopier et compléter le code de la méthode `taille` de la classe `File`.

On souhaite écrire un petit jeu de tests pour vérifier que cette structure de file est bien codée.

11. Écrire, à l'aide des méthodes de la classe `File`, une séquence d'instructions qui :
    - crée une file vide ;
    - ajoute `1` puis `2` puis `3` dans cette file ;
    - vérifie qu'elle est de la longueur attendue à l'aide d'un `assert` ;
    - défile un à un les trois éléments et vérifie, à l'aide d'`assert`, qu'ils sortent dans le bon ordre ;
    - vérifie enfin que la file est bien vide.

## PARTIE C : RÉSOLUTION DU JEU

Afin de résoudre ce jeu, on va énumérer tous les chemins non prolongeables, puis en extraire les chemins les plus longs.

La recherche de tous les chemins non prolongeables depuis un sommet se fait à l'aide d'un algorithme reposant sur une file de chemins :

- on crée une liste de chemins non prolongeables initialement vide ;
- on crée une file de chemins à prolonger initialement vide ;
- pour chaque sommet `s` du jeu, on y enfile le chemin `[s]` ;
- tant que cette file est non vide :
    - on retire un chemin de la file ;
    - on considère le dernier sommet de ce chemin ;
    - pour chaque voisin (diviseur ou multiple), de ce dernier sommet :
- si le voisin n'est pas déjà dans le chemin
    - on ajoute à la file le nouveau chemin obtenu en prolongeant le chemin considéré par ce voisin, – s'il n'a pas été possible de prolonger le chemin :
- on ajoute ce chemin à la liste des chemins non prolongeables ;
- on renvoie finalement liste des chemins non prolongeables .

Voici le code incomplet de cette fonction en Python :

```
1  def rechercher_chemins(jeu):
2      chemins_np = []
3      f = File()
4      for sommet in jeu :
5          f.enfiler([sommet])
6      while ...:
7          chemin = ...
8          dernier = ...
9          voisins = dernier.diviseurs + dernier.multiples
10         prolongeable = ...
11         for voisin in voisins:
12             if voisin not in chemin:
13                 prolongeable = ...
14                 f.enfiler(...)
15         if not prolongeable:
16             chemins_np.append(chemin)
17     return chemins_np
```

12. Recopier et compléter les lignes 6, 7, 8, 10, 13 et 14 du code ci-dessus de la fonction `rechercher_chemins`.

Un chemin est une liste de sommets (une liste d'instances de la classe `Sommet`), mais pour l'affichage, il est intéressant de considérer la liste des entiers qu'ils représentent.

13. Écrire une fonction `valeurs_chemin` qui prend en paramètre `chemin` une liste de sommets et renvoie la liste de leurs valeurs.

14. Recopier et compléter, à l'aide des fonctions `rechercher_chemins` et `chemin_valeur`, les lignes de code suivantes qui permettent d'obtenir tous les chemins non prolongeables du jeu à 9 entiers représentés par la liste des valeurs de leurs sommets.

```
1 chemins = ...
2 for chemin in ...:
3     chemins.append( ... )
```

Dans la liste `chemins` nous avons désormais l'ensemble de tous les chemins (non prolongeables) possibles dans le jeu sous la forme de listes d'entiers. Notre objectif est de trouver le ou les plus longs chemin(s) dans cette liste (il peut y avoir plusieurs chemins dont la longueur est maximale).

15. Écrire le code de la fonction `extraire_plus_longs_chemins` qui prend en paramètre une liste de chemins `L_chemins` et qui renvoie la liste des chemins de `L` de longueur maximale.

Exemple de fonctionnement de cette fonction :

```
>>> extraire_plus_longs_chemins([[1,3,6,2,4,8]])
[1,3,6,2,4,8]
>>> extraire_plus_longs_chemins([[1,7],[3,1,5],[4,1,7]])
[[3,1,5],[4,1,7]]
```

Nous avons testé l'ensemble de notre programme sur différentes tailles de jeu. Voici le nombre de chemins (non prolongeables) obtenus en fonction du nombre de sommets dans le graphe du jeu :

| Nombre de sommets | `len(L_chemins)` |
|---|---|
| 9 | 377 |
| 10 | 800 |
| 11 | 925 |
| 12 | 5279 |
| 13 | 5935 |
| 14 | 9896 |

16. Dire si l'on peut, à l'aide de notre programme, résoudre le jeu pour un nombre de sommets aussi grand que l'on souhaite. Justifier.