Bac Général Maths + NSI (Mathématiques et Numérique et sciences informatiques)

SUJET D'EXAMEN (SESSION OFFICIELLE)

Numérique et sciences informatiques - Session 2025

Sujet de l'examen national tel que publié pour la session indiquée. Source : sujets nationaux publiés par le ministère chargé de l'examen et les services académiques. Document mis à disposition par FormaV pour la révision.

Toutes les annales et leurs corrigés : https://bac-maths-nsi.fr/annales-sujets-et-corriges

BACCALAURÉAT GÉNÉRAL

ÉPREUVE D'ENSEIGNEMENT DE SPÉCIALITÉ

SESSION 2025

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 16 pages numérotées de 1/16 à 16/16.

Le sujet est composé de trois exercices indépendants.

Le candidat traite les trois exercices.

Exercice 1 (6 points)

Cet exercice porte sur les bases de données relationnelles et les requêtes SQL.

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` et `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`.

La ville de Bois-Plage a décidé d'organiser, pendant un mois de juillet, un tournoi sportif de volley-ball par équipes de 4. Elle met à disposition des personnes intéressées un site d'inscription en ligne qui utilise un système de gestion de base de données.

Le schéma de la base de données utilisée est donné ci-dessous, en figure 1. Sur ce schéma, les clés primaires ont été soulignées et les clés étrangères indiquées par un croisillon (symbole #).

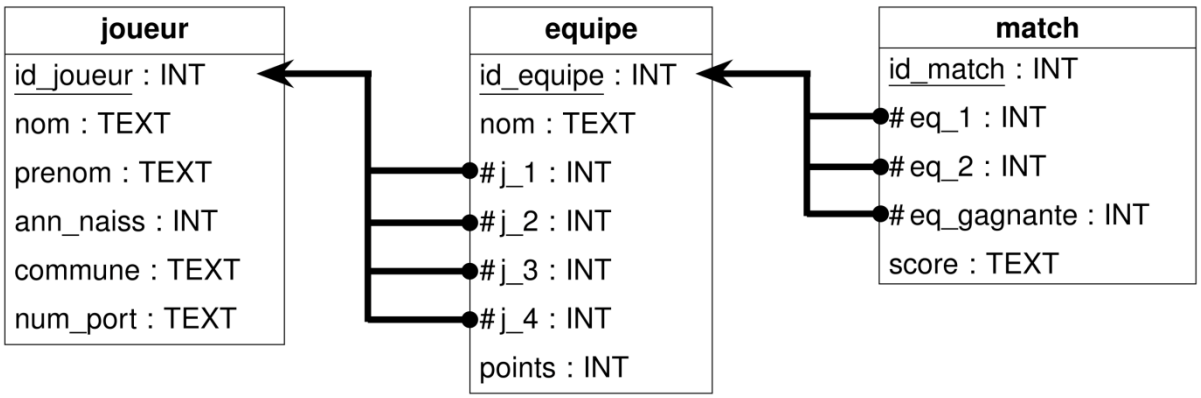

Figure 1. Schéma de la base de données

À la clôture des inscriptions, de nombreuses équipes sont inscrites.

Sur la page suivante sont présentés des extraits des tables `joueur` et `equipe` obtenues à l'issue de la phase d'inscription.

joueur					
id_joueur	nom	prenom	ann_naiss	commune	num_port
25	Leclerc	Océane	2008	Bois-Plage	0660358945
26	Renault	Henri	1971	Guilland	0625597427
27	Desousa	Laure	1980	Bois-Plage	0746881113
28	Hernand	Yves	1986	Lebrundan	0739401689
29	Giraud	Brigitte	1972	Saint-Adrien	0651936319
30	Barbier	Laure	1979	Bois-Plage	0787028125

equipe						
id_equipe	nom	j_1	j_2	j_3	j_4	points
8	Les Mr Freeze	7	12	5	33	0
9	Tagadas Winners	45	23	67	65	0
10	Volley Warriors	25	27	30	35	0
11	Les Piafs	37	32	41	28	0

1. Expliquer, dans les relations précédentes, le rôle des clés primaires.
2. Expliquer quelles données n'auraient pas pu être stockées dans la table `match` si le champ `id_match` n'avait pas été introduit dans cette table.
3. Donner le résultat de la requête suivante en l'appliquant à l'extrait de la table `joueur` donné dans l'énoncé.

```
SELECT prenom FROM joueur WHERE ann_naiss < 1985
```

4. Modifier la requête précédente afin d'éviter les éventuels doublons.
5. Écrire une requête SQL permettant d'obtenir tous les noms, années de naissance et numéros de téléphone portable des personnes qui habitent à Bois-Plage.

L’organisateur souhaite obtenir l’identité du premier joueur de l’équipe “les Kangourous”. L’équipe “Les Kangourous” n’apparaît pas dans l’extrait.

6. Écrire une requête SQL permettant d’obtenir le nom et le prénom du joueur `j_1` de l’équipe “Les Kangourous”.

L’équipe “Volley Warriors” a terminé le tournoi avec un total de 5 points.

7. Écrire une requête SQL permettant de mettre à jour la table `equipe` avec le nombre de points gagnés par l’équipe “Volley Warriors”.

8. Écrire une requête SQL permettant de supprimer de la table `joueur` le joueur ayant pour identifiant le numéro 35.

À la clôture du tournoi, la table `match` est totalement complétée. Un extrait de cette table est donné ci-dessous.

match				
id_match	eq_1	eq_2	eq_gagnante	score
32	3	8	8	25-20
33	3	9	3	25-15
34	3	10	10	25-7

9. Écrire une requête SQL permettant d’obtenir la liste des identifiants de matchs auxquels a participé l’équipe ayant pour identifiant 12.

10. Écrire une requête SQL permettant d’obtenir la liste des identifiants des matchs pour lesquels le joueur 1 de l’équipe 1 du match vient de la commune de Bois-Plage.

11. Écrire une requête SQL permettant d’obtenir la liste, classée par ordre alphabétique, des noms et prénoms des joueurs ayant gagné au moins un match en tant que joueur 1 de l’équipe 1 du match.

Exercice 2 (6 points)

Cet exercice porte sur les listes, les dictionnaires, les fonctions et la récursivité.

Nous souhaitons créer en langage Python un dictionnaire contenant un grand nombre de mots de façon à ce qu'une recherche dans ce dictionnaire soit la plus rapide possible.

Pour cela nous allons créer des groupes de mots, chaque groupe sera une liste Python associée à une clé unique dans le dictionnaire.

Voici un extrait du dictionnaire que nous souhaitons créer (les `...` indiquent des éléments non listés dans cet extrait) :

```
d = {
    44  : ['ABAISSEMENT', 'ADMINISTRATEUR', ..., 'VERSETS'],
    74  : ['ABAISSER', 'ABLATION', ..., 'TROU'],
    243 : ['ABANDON', 'ALLEGRETTO', ..., 'ZIP'],
    36  : ['ABANDONNANT', 'ABOLITIONNISTE', ..., 'VOULAIT'],
    134 : ['ABANDONNE', 'AGNOSTICISME', ..., 'VOIES'],
    40  : ['ABANDONNENT', 'ACCOUCHEUSE', ..., 'YACK'],
    ...
}
```

Chaque clé sera un nombre entier positif et chaque valeur sera une liste de mots.

Partie A

Pour générer ces valeurs de clés, nous utiliserons une fonction dite de *hachage*, c'est-à-dire une fonction qui pour un mot donné calculera la clé qui lui sera associée. Nous choisissons une fonction de *hachage* simple qui consiste à additionner sur un octet les codes ASCII de chaque lettre de ce mot.

Nous n'utiliserons que des lettres majuscules de l'alphabet, nous rappelons ici le code ASCII (en hexadécimal, c'est-à-dire en base 16) de chacune de ces lettres. Chaque valeur en hexadécimal est notée avec le préfixe `0x`, par exemple `0x15` correspond en décimal à $1 \times 16 + 5 \times 1 = 21$.

Table des codes ASCII					
'A' : 0x41	'F' : 0x46	'K' : 0x4B	'P' : 0x50	'U' : 0x55	'Z' : 0x5A
'B' : 0x42	'G' : 0x47	'L' : 0x4C	'Q' : 0x51	'V' : 0x56	
'C' : 0x43	'H' : 0x48	'M' : 0x4D	'R' : 0x52	'W' : 0x57	
'D' : 0x44	'I' : 0x49	'N' : 0x4E	'S' : 0x53	'X' : 0x58	
'E' : 0x45	'J' : 0x4A	'O' : 0x4F	'T' : 0x54	'Y' : 0x59	

Ainsi le mot `'NSI'` aura pour clé `0xEA` puisque : `0x4E + 0x53 + 0x49 = 0xEA`, ou `234` en décimal.

Si la somme des codes ASCII ne tient pas sur un octet, seul l'octet de poids faible est conservé.

Par exemple : la somme des codes `ASCII` des lettres du mot `'ADMINISTRATEUR'` est de `0x42C`, la clé qui lui sera associée sera donc `0x2C`, égale à `44` en décimal.

1. Trouver la valeur exprimée en hexadécimal de la clé qui sera associée au mot `'EW'`.

2. Comparer, sans les calculer, les valeurs des clés qui seront associées aux mots `'SAC'` et `'CAS'`. Justifier la réponse.

Voici le code de la fonction qui calcule cette clé pour un mot donné (nous rappelons que la fonction `ord` renvoie le code ASCII d'un caractère) :

```
1 def code_hachage(mot):
2     somme = ...
3     for caractere in ...:
4         somme = ...
5     return somme % 0x100
```

3. Recopier et compléter les lignes 2, 3 et 4 de la fonction `code_hachage`.

4. Expliquer l'expression `somme % 0x100` dans le code ci-dessus et donner les valeurs possibles pour la clé.

Partie B

Les listes de mots associées à chaque clé sont rangées dans l'ordre alphabétique, ce qui facilitera la recherche d'un mot par la suite.

Nous allons maintenant voir l'écriture d'une fonction permettant l'ajout d'un mot dans une liste de mots en maintenant un ordre alphabétique dans cette liste.

Nous rappelons qu'en langage Python la comparaison de chaînes de caractères est possible à l'aide des opérateurs classiques de comparaison : <, >, ==, >= et <= . Ces opérateurs utilisent l'ordre alphabétique.

Par exemple, l'expression booléenne `'ANNEE' < 'BATEAU'` vaut `True` puisque le mot `'ANNEE'` est placé avant le mot `'BATEAU'` dans l'ordre alphabétique.

Voici le code d'une fonction `ajouter_mot_liste(liste, mot)` qui a pour paramètres `liste`, une liste de chaînes de caractères (ce sont les mots) et `mot`, une chaîne de caractères correspondant au mot que l'on souhaite ajouter à `liste`. Cette fonction modifie `liste` en y ajoutant `mot` selon l'ordre alphabétique. Elle renvoie de plus `liste` après cet ajout.

```
 1 def ajouter_mot_liste(liste, mot):
 2     i = 0
 3     while i < len(liste):
 4         if mot < liste[i]:
 5             # La méthode "insert" permet d'ajouter un
 6             # élément à un indice donné dans "liste",
 7             # les éléments suivants sont décalés.
 8             liste.insert(i, mot)
 9             return liste
10         i = i + 1
11     # La méthode "append" permet d'ajouter un nouvel
12     # élément à la fin de "liste".
13     liste.append(mot)
14     return liste
```

Voici des exemples d'utilisation de cette fonction :

```
>>> ajouter_mot_liste([],'NSI')
['NSI']
>>> ajouter_mot_liste(['NSI'],'PYTHON')
['NSI','PYTHON']
>>> ajouter_mot_liste(['NSI','PYTHON'],'OBJET')
['NSI','OBJET','PYTHON']
>>> ajouter_mot_liste(['NSI','OBJET','PYTHON'],'RAM')
['NSI','OBJET','PYTHON','RAM']
```

5. Déterminer, dans le pire cas, l'ordre de grandeur du nombre de comparaisons entre chaînes de caractères effectuées par un appel à `ajouter_mot_liste`. On exprimera le résultat en fonction de n, le nombre de mots présents dans `liste`.

Nous souhaitons écrire une fonction qui permette l'ajout d'un mot dans le dictionnaire décrit au début de l'exercice (qui associe à chaque clé entière possible une liste de mots).

6. Rappeler avec quelle expression Python on peut tester si une clé c est déjà présente dans un dictionnaire `dico`.

7. Écrire un code pour la fonction `ajouter_mot_dict(dict_mots, mot)` qui a pour paramètres `dict_mots`, un dictionnaire qui associe à chaque clé entière possible une liste de mots, et `mot` le mot à ajouter dans `dict_mots`, et qui réalise cet ajout (mais ne renvoie aucune valeur). On utilisera les fonctions `code_hachage` et `ajouter_mot_liste`.

Partie C

Nous allons maintenant voir comment faire la recherche d'un mot dans notre dictionnaire.

Nous souhaitons une recherche rapide et nous allons profiter du fait que les mots sont classés dans l'ordre alphabétique dans les listes du dictionnaire.

Nous prendrons une approche dichotomique pour écrire une fonction `est_present(liste, mot, debut, fin)` qui a pour paramètres :

- `liste`, une liste de mots classés dans l'ordre alphabétique ;
- `mot`, le mot à rechercher ;
- `debut` et `fin`, les indices entre lesquels la recherche se fait dans `liste`.

Cette fonction renverra `True` si `liste` contient `mot` entre les indices `debut` (inclus) et `fin` (exclus), et renverra `False` sinon.

Voici son code :

```
 1 def est_present(liste, mot, debut, fin):
 2     if debut > fin:
 3         return False
 4     milieu = (debut + fin) // 2
 5     if liste[milieu] > mot:
 6         return est_present(liste, mot, 0, milieu - 1)
 7     elif liste[milieu] < mot:
 8         return est_present(liste, mot, milieu + 1, fin)
 9     else:
10         return True
```

Nous testons cette fonction ainsi :

```
>>> liste_mots = ['FONCTION', 'NSI', 'PYTHON', 'OBJET', 'RAM']
>>> est_present(liste_mots, 'NSI', 0, len(liste_mots))
True
```

8. Donner, lors de l'exécution de l'exemple précédent, les valeurs des paramètres `debut` et `fin` prises lors de chaque appel de la fonction `est_present`.

9. Expliquer pourquoi la méthode dichotomique permet d'effectuer, en général, moins d'opérations qu'une recherche simple qui consisterait à comparer un par un les mots de la liste avec le mot cherché.

10. Donner l'ordre de grandeur du nombre de comparaisons de mots effectuées par la méthode dichotomique sur une liste de longueur n ?

11. Écrire un code pour la fonction `mot_present(dict_mots, mot)` qui a pour paramètres `dict_mots` , un dictionnaire qui associe à chaque clé entière possible une liste de mots, et `mot`, le mot à rechercher dans `dict_mots`, et qui renvoie `True` si le mot est présent et `False` sinon. On utilisera les fonctions `code_hachage` et `est_present`.

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.

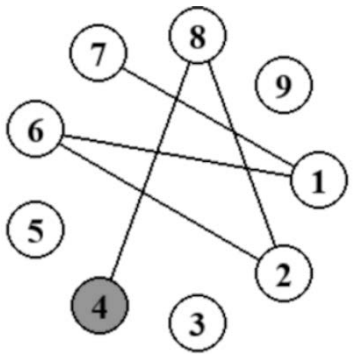

Figure 1. Un chemin de longueur 5 depuis le nombre 4

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.

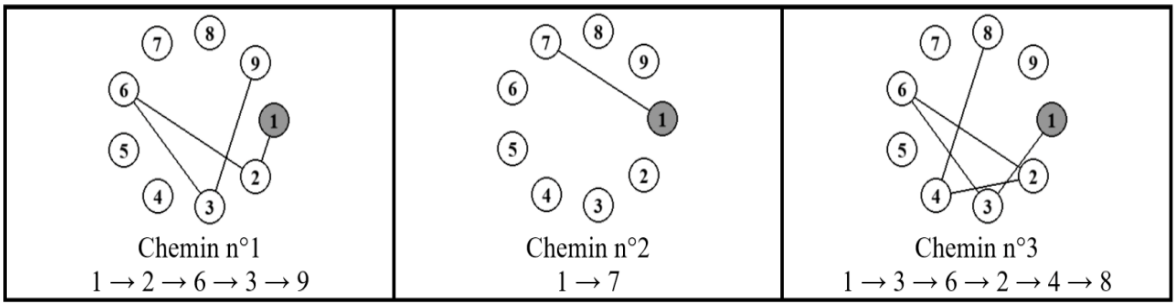

Figure 2. Trois chemins non prolongeables possibles en partant du nombre 1

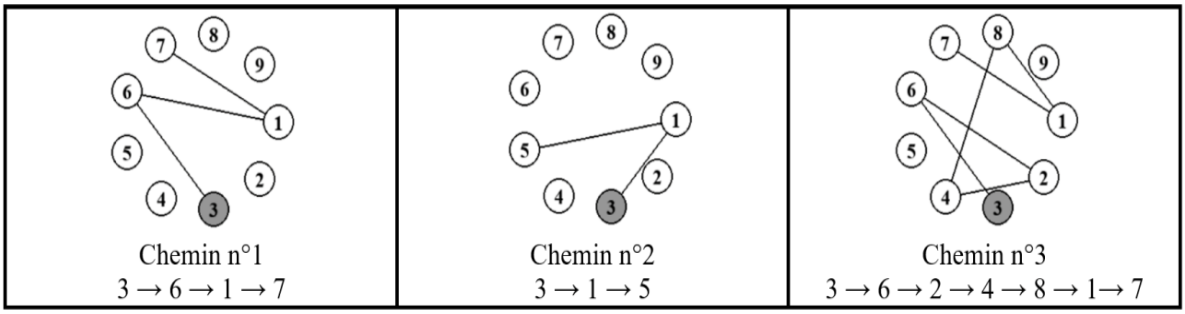

Figure 3. Trois chemins non prolongeables possibles en partant du nombre 3.

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

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.

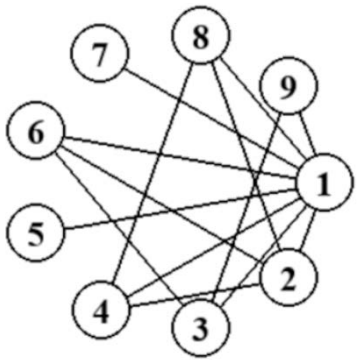

Figure 4. Le graphe du jeu à 9 entiers

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 n est une liste qui contiendra tous les objets de type `Sommet` représentant un diviseur (respectivement un multiple) de l'entier n.

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 ci-dessous 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 ci-dessous

```
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`.

```
1 l_div_3 = ...
2 l_mult_3 = ...
3 assert 1 in l_div_3
4 assert ... in ...
5 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.