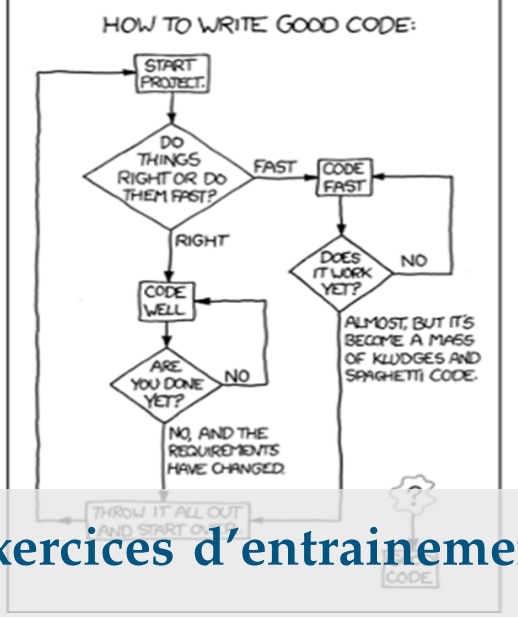

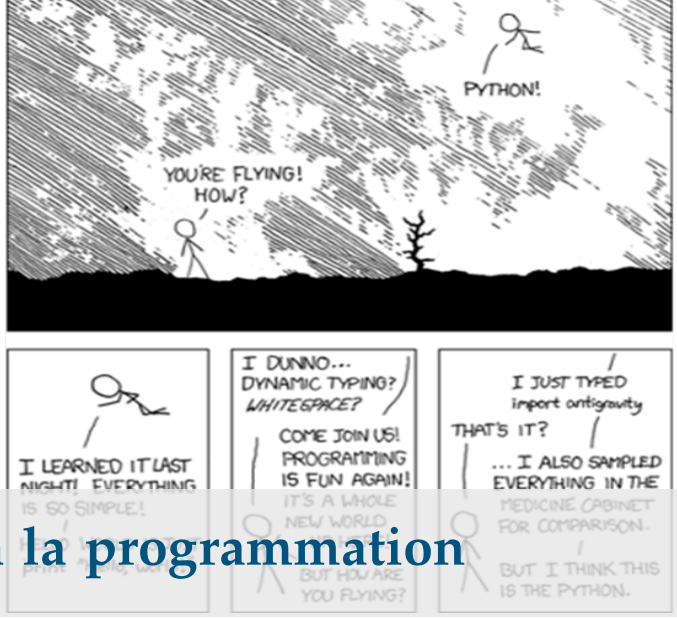

Exercices d'entrainement à la programmation

0.1 Exercices boucle `for`

0.1.1 Niveau 1

1. Écrivez un programme qui affiche les nombres de 1 à 10 en utilisant une boucle `for`.
2. Écrivez un programme qui affiche tous les nombres pairs de 1 à 20.
3. Créez un programme qui affiche les carrés des nombres de 1 à 10.
4. Écrivez un programme qui affiche les éléments d'une liste de fruits `['pomme', 'banane', 'orange', 'raisin']`.
5. Écrivez un programme qui calcule et affiche la somme des nombres de 1 à 100.
6. Créez un programme qui affiche les lettres de l'alphabet en utilisant une boucle `for`.
7. Écrivez un programme qui prend une liste de nombres et affiche chaque nombre multiplié par 2.
8. Écrivez un programme qui affiche tous les éléments d'une liste de noms en commençant par la première lettre de chaque nom en majuscule.
9. Créez un programme qui affiche les 10 premiers multiples de 3.
10. Écrivez un programme qui affiche les nombres de 10 à 1 dans l'ordre décroissant.
11. Écrivez un programme qui affiche la table de multiplication de 7 (de 1 à 10).
12. Créez un programme qui demande à l'utilisateur un nombre entier, puis affiche les `n` premiers nombres impairs, où `n` est l'entier donné.
13. Écrivez un programme qui prend une phrase et affiche chaque mot de cette phrase séparément.
14. Créez un programme qui demande à l'utilisateur une chaîne de caractères et qui affiche chaque caractère de la chaîne individuellement.
15. Écrivez un programme qui utilise une boucle `for` pour compter le nombre de voyelles dans une phrase donnée.
16. Créez un programme qui utilise une boucle `for` pour calculer la factorielle d'un nombre donné par l'utilisateur.
17. Écrivez un programme qui prend une liste de notes (entiers) et calcule la moyenne de ces notes.
18. Créez un programme qui demande à l'utilisateur un nombre et affiche les diviseurs de ce nombre.
19. Écrivez un programme qui affiche les nombres de 1 à 50, mais pour les multiples de 3 affiche "Fizz", pour les multiples de 5 affiche "Buzz", et pour les multiples

de 3 et 5 affiche "FizzBuzz".

20. Créez un programme qui affiche les nombres de la suite de Fibonacci jusqu'au 10^{e} terme.

0.1.2 Niveau 2

1. Écrivez un programme qui prend une liste de mots et affiche le mot le plus long de cette liste. Utilisez une boucle `for` pour parcourir la liste.
2. Écrivez un programme qui affiche les nombres premiers de 1 à 100. Utilisez une boucle `for` imbriquée pour vérifier si chaque nombre est divisible uniquement par 1 et par lui-même.
3. Créez un programme qui prend un mot et vérifie s'il est un palindrome (mot qui se lit de la même manière à l'envers). Utilisez une boucle `for` pour comparer les caractères.
4. Écrivez un programme qui calcule le produit scalaire de deux listes d'entiers de même longueur. Par exemple, si `A = [1, 2, 3]` et `B = [4, 5, 6]`, le produit scalaire est `1*4 + 2*5 + 3*6 = 32`.
5. Créez un programme qui génère une pyramide de nombres. Par exemple, si l'utilisateur entre `4`, le programme doit afficher :

```
1
22
333
4444
```

6. Écrivez un programme qui utilise une boucle `for` pour trier une liste d'entiers en utilisant l'algorithme de tri par sélection. Ne pas utiliser de fonctions de tri intégrées.
7. Créez un programme qui prend une liste de phrases et affiche les mots de chaque phrase qui contiennent exactement 5 lettres. Utilisez une boucle `for` pour parcourir chaque mot.
8. Écrivez un programme qui utilise une boucle `for` pour compter le nombre de fois où chaque lettre apparaît dans une phrase donnée. Affichez le résultat sous forme de dictionnaire où chaque lettre est une clé.
9. Créez un programme qui prend une liste de listes (par exemple `[[1, 2], [3, 4], [5, 6]]`) et calcule la somme de tous les éléments. Utilisez des boucles `for` imbriquées pour parcourir la liste principale et les sous-listes.
10. Écrivez un programme qui génère les `n` premiers termes de la séquence de Fibonacci et les stocke dans une liste. Utilisez une boucle `for` pour générer les termes et afficher la liste finale.

0.1.3 Niveau 3

1. Créez un programme qui prend une liste de mots et détermine, pour chaque mot, si celui-ci est un palindrome. Le programme doit afficher les mots qui sont des palindromes.
2. Écrivez un programme qui calcule la somme de tous les nombres premiers inférieurs à un nombre donné par l'utilisateur. Utilisez des boucles `for` imbriquées pour vérifier si chaque nombre est premier.
3. Créez un programme qui prend deux chaînes de caractères et trouve les lettres communes entre les deux chaînes, sans répétition. Utilisez une boucle `for` pour comparer les lettres et construisez le résultat dans une liste ou une chaîne.

4. Écrivez un programme qui génère la suite de Collatz pour un nombre donné. La suite de Collatz est définie par : si le nombre est pair, le diviser par `2` ; s'il est impair, le multiplier par 3 et ajouter 1. La suite continue jusqu'à ce que le nombre atteigne 1.
5. Créez un programme qui prend une liste de listes représentant une matrice carrée et calcule la somme des éléments au-dessus de la diagonale principale.
6. Écrivez un programme qui génère les premiers `n` nombres de la suite de Lucas, une variante de la suite de Fibonacci avec les termes initiaux `2` et `1`.
7. Créez un programme qui prend une phrase et affiche chaque mot avec ses lettres en ordre alphabétique. Par exemple, pour "bonjour tout le monde", le programme devrait retourner "bjnoru ottu el ddmeno".
8. Écrivez un programme qui trouve tous les nombres parfaits inférieurs à un nombre donné. Un nombre parfait est un nombre égal à la somme de ses diviseurs propres (comme 6, qui est 1 + 2 + 3).
9. Créez un programme qui prend une liste de nombres et calcule le produit de tous les éléments pairs, tout en additionnant les éléments impairs. Affichez le produit des pairs et la somme des impairs.
10. Écrivez un programme qui génère les `n` premières lignes du triangle de Pascal et les affiche sous forme de liste de listes. Utilisez une boucle `for` pour construire chaque ligne.

0.1.4 Niveau 4

0.1.5 Niveau 4

1. Écrivez un programme qui trie une liste d'entiers en utilisant l'algorithme de tri par insertion. Ne pas utiliser les fonctions de tri intégrées de Python, mais seulement des boucles `for` et des comparaisons.
2. Créez un programme qui prend une matrice (liste de listes) et calcule la somme des éléments de la diagonale principale (du coin supérieur gauche au coin inférieur droit). Utilisez une boucle `for` pour accéder aux éléments de la diagonale.
3. Écrivez un programme qui génère tous les sous-ensembles possibles d'une liste donnée, sans utiliser de bibliothèques ou de fonctions prédéfinies pour le faire. Par exemple, pour `[1, 2, 3]`, le programme devrait générer `[[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]]`.
4. Créez un programme qui vérifie si deux chaînes de caractères sont des anagrammes (contiennent les mêmes lettres en quantités identiques, sans ordre particulier). Utilisez des boucles `for` pour compter et comparer les lettres de chaque chaîne.
5. Écrivez un programme qui implémente l'algorithme de tri rapide (quick sort) en utilisant des boucles `for` et des fonctions récursives. Ne pas utiliser les fonctions de tri intégrées.
6. Créez un programme qui prend une liste de chaînes de caractères et regroupe celles qui sont des anagrammes. Par exemple, pour `['rat', 'tar', 'art', 'bat', 'tab']`, le programme devrait retourner `[['rat', 'tar', 'art'], ['bat', 'tab']]`.
7. Écrivez un programme qui utilise une boucle `for` pour générer les nombres de la suite de Fibonacci jusqu'à ce qu'un terme dépasse une valeur donnée par l'utilisateur. Affichez chaque terme généré.
8. Créez un programme qui, donné un nombre entier positif, vérifie si ce nombre est un nombre parfait (c'est-à-dire égal à la somme de ses diviseurs propres). Par exemple, 6 est parfait car `1 + 2 + 3 = 6`.

9. Écrivez un programme qui prend une matrice carrée (liste de listes) et vérifie si elle est symétrique (égale à sa transposée). Utilisez une boucle `for` pour comparer les éléments de la matrice.
10. Créez un programme qui génère le triangle de Pascal jusqu'à une hauteur donnée. Utilisez une boucle `for` pour calculer chaque ligne du triangle. Par exemple, pour une hauteur de 5, le triangle doit ressembler à :

```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
```

0.2 Exercices boucle `while`

0.2.1 Niveau 1

Exercices - Boucles for

0.2.1 Niveau débutant

1. **Exercice 1**
   Utilisez une boucle `while` pour afficher les nombres de 1 à 10.
2. **Exercice 2**
   Créez une boucle `while` qui affiche les nombres pairs de 2 à 20.
3. **Exercice 3**
   Utilisez une boucle `while` pour afficher les multiples de 3, de 3 à 30 inclus.
4. **Exercice 4**
   Écrivez une boucle `while` qui affiche tous les nombres de 10 à 1, dans l'ordre décroissant.
5. **Exercice 5**
   Utilisez une boucle `while` pour calculer et afficher la somme des nombres de 1 à 50.
6. **Exercice 6**
   Utilisez une boucle `while` pour calculer et afficher la somme des nombres pairs entre 1 et 100.
7. **Exercice 7**
   Écrivez une boucle `while` pour calculer et afficher le produit des nombres de 1 à 10.
8. **Exercice 8**
   Créez une boucle `while` qui calcule la puissance de 2, de 2^1 à 2^{10}, et affiche chaque résultat.
9. **Exercice 9**
   Utilisez une boucle `while` pour afficher la suite de Fibonacci, jusqu'au $10^{\text{ème}}$ terme.
10. **Exercice 10**
    Utilisez une boucle `while` pour afficher les nombres de 1 à 100. Si un nombre est multiple de 3, affichez `Fizz` à la place, et si un nombre est multiple de 5, affichez `Buzz`. Si un nombre est multiple de 3 et de 5, affichez `FizzBuzz`.

0.2.2 Niveau 2

1. **Calcul de la somme des entiers**
   Écrire un programme qui demande à l'utilisateur un nombre entier positif n et qui utilise une boucle `while` pour calculer la somme des entiers de 1 à n inclus.
2. **Trouver le plus petit multiple**
   Demandez à l'utilisateur de saisir deux nombres entiers positifs, a et b. Utilisez une boucle `while` pour trouver le plus petit multiple commun de a et b.
3. **Calcul de la factorielle**
   Écrire un programme qui demande à l'utilisateur un nombre entier positif n et utilise une boucle `while` pour calculer la factorielle de n, notée $n!$. Rappel : $n! = 1 \times 2 \times 3 \times \cdots \times n$.
4. **Inversion d'un nombre**
   Écrire un programme qui demande un nombre entier positif à l'utilisateur et utilise une boucle `while` pour inverser ses chiffres. Par exemple, si l'utilisateur entre 1234, le programme doit afficher 4321.
5. **Deviner un nombre**
   Écrire un programme qui génère un nombre aléatoire entre 1 et 100. Le programme doit permettre à l'utilisateur de deviner le nombre en entrant des propositions successives. Utilisez une boucle `while` pour permettre à l'utilisateur de deviner jusqu'à ce qu'il trouve le bon nombre.
6. **Nombre de chiffres**
   Écrire un programme qui demande à l'utilisateur un nombre entier positif et utilise une boucle `while` pour déterminer le nombre de chiffres dans ce nombre. Par exemple, si l'utilisateur entre 1234, le programme doit afficher 4.
7. **Suite de Fibonacci**
   Écrire un programme qui affiche les n premiers termes de la suite de Fibonacci, où n est un nombre entier positif donné par l'utilisateur. Utilisez une boucle `while` pour calculer les termes. Rappel : La suite de Fibonacci commence par 0 et 1, et chaque terme suivant est la somme des deux précédents.
8. **Syracuse (ou Suite de Collatz)**
   Demandez à l'utilisateur un nombre entier positif n. Écrivez un programme qui utilise une boucle `while` pour générer la suite de Syracuse de n, où chaque terme est calculé de la manière suivante :

   $$n = \begin{cases} n/2 & \text{si } n \text{ est pair} \\ 3n+1 & \text{si } n \text{ est impair} \end{cases}$$

   La suite continue jusqu'à ce que n atteigne 1.
9. **Produit des entiers pairs**
   Écrire un programme qui demande un nombre entier positif n et utilise une boucle `while` pour calculer le produit des entiers pairs de 2 jusqu'à n. Si n est impair, ignorez-le.
10. **Jeu du palindrome**
    Écrire un programme qui demande à l'utilisateur d'entrer un mot et utilise une boucle `while` pour inverser le mot. Le programme doit ensuite indiquer si le mot est un palindrome (un mot qui se lit de la même manière dans les deux sens).

0.2.3 Niveau 3

1. **Nombre parfait**
   Un nombre parfait est un entier positif égal à la somme de ses diviseurs positifs propres (autres que lui-même). Par exemple, 6 est un nombre parfait car

$6 = 1 + 2 + 3$. Écrivez un programme qui demande un entier positif n et utilise une boucle `while` pour déterminer si n est un nombre parfait.

2. **Trouver les nombres premiers inférieurs à n**
   Demandez à l'utilisateur un nombre entier positif n. Écrivez un programme qui utilise une boucle `while` pour afficher tous les nombres premiers inférieurs à n. Assurez-vous d'optimiser le programme pour réduire le nombre de calculs nécessaires.
3. **Approximation de π**
   Écrire un programme qui utilise une boucle `while` pour approximer la valeur de π en utilisant la série de Leibniz :

   $$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots\right)$$

   Continuez jusqu'à ce que la différence entre deux itérations successives soit inférieure à 10^{-6}.
4. **Décomposition en facteurs premiers**
   Écrire un programme qui demande à l'utilisateur un entier positif n et utilise une boucle `while` pour décomposer n en facteurs premiers. Par exemple, pour $n = 60$, le programme devrait afficher $2 \times 2 \times 3 \times 5$.
5. **Nombre de Kaprekar**
   Un nombre de Kaprekar est un nombre n tel que n^2 peut être décomposé en deux parties dont la somme est égale à n. Par exemple, 45 est un nombre de Kaprekar car $45^2 = 2025$ et $20 + 25 = 45$. Écrire un programme qui vérifie si un nombre donné par l'utilisateur est un nombre de Kaprekar.
6. **Conjecture de Syracuse**
   Écrire un programme qui demande à l'utilisateur un entier positif n et utilise une boucle `while` pour vérifier la conjecture de Syracuse, aussi connue sous le nom de conjecture de Collatz. Comptez le nombre d'étapes nécessaires pour que n atteigne 1.
7. **Trouver les nombres d'Armstrong**
   Un nombre d'Armstrong est un nombre qui est égal à la somme de ses chiffres chacun élevé à la puissance du nombre de chiffres. Par exemple, 153 est un nombre d'Armstrong car $153 = 1^3 + 5^3 + 3^3$. Écrire un programme qui demande un nombre entier n et affiche tous les nombres d'Armstrong inférieurs ou égaux à n.
8. **Racine carrée par la méthode de Newton**
   Écrire un programme qui demande un nombre positif x et utilise la méthode de Newton pour calculer la racine carrée de x. Continuez les itérations jusqu'à ce que la différence entre deux approximations successives soit inférieure à 10^{-6}.
9. **Ensemble de nombres parfaits inférieurs à n**
   Demandez à l'utilisateur un nombre entier positif n. Écrivez un programme qui utilise une boucle `while` pour trouver tous les nombres parfaits inférieurs à n. Les nombres parfaits sont rares, donc le programme devra être efficace.
10. **Détection des cycles dans une suite**
    Demandez à l'utilisateur de saisir un entier n et définissez la suite suivante :

    $$n = \begin{cases} n/2 & \text{si } n \text{ est pair} \\ 3n + 1 & \text{si } n \text{ est impair} \end{cases}$$

    Utilisez une boucle `while` pour vérifier si la suite entre dans un cycle. Affichez le cycle trouvé.

0.3 Exercices sur les `list`

0.3.1 Niveau 1

1. **Création d'une liste**
   Écrivez un programme qui demande à l'utilisateur de saisir 5 nombres et qui les stocke dans une liste. Affichez ensuite la liste complète.
2. **Accéder aux éléments de la liste**
   Écrivez un programme qui demande à l'utilisateur de saisir une liste de 5 éléments. Affichez le premier et le dernier élément de la liste.
3. **Trouver la longueur d'une liste**
   Écrivez un programme qui demande à l'utilisateur de saisir une liste d'éléments et affiche la longueur de la liste.
4. **Additionner les éléments d'une liste**
   Écrivez un programme qui demande à l'utilisateur de saisir une liste de nombres et calcule la somme des éléments de la liste.
5. **Trouver le maximum et le minimum**
   Écrivez un programme qui demande à l'utilisateur de saisir une liste de nombres et affiche le plus grand et le plus petit nombre de la liste.
6. **Inverser une liste**
   Écrivez un programme qui demande à l'utilisateur de saisir une liste d'éléments et affiche la liste inversée.
7. **Vérifier la présence d'un élément**
   Écrivez un programme qui demande à l'utilisateur de saisir une liste d'éléments et un élément à rechercher. Indiquez si l'élément est présent ou non dans la liste.
8. **Compter les occurrences d'un élément**
   Écrivez un programme qui demande à l'utilisateur de saisir une liste d'éléments et un élément. Comptez le nombre de fois que cet élément apparaît dans la liste.
9. **Ajouter un élément à la fin**
   Écrivez un programme qui demande à l'utilisateur de saisir une liste d'éléments et un nouvel élément. Ajoutez cet élément à la fin de la liste et affichez la liste mise à jour.
10. **Insérer un élément à une position donnée**
    Écrivez un programme qui demande à l'utilisateur de saisir une liste, un élément, et une position. Insérez l'élément à la position spécifiée dans la liste et affichez la liste mise à jour.
11. **Supprimer un élément donné**
    Écrivez un programme qui demande à l'utilisateur de saisir une liste d'éléments et un élément à supprimer. Retirez cet élément de la liste et affichez la liste mise à jour.
12. **Supprimer l'élément à une position donnée**
    Écrivez un programme qui demande à l'utilisateur de saisir une liste et une position. Supprimez l'élément à cette position et affichez la liste mise à jour.
13. **Trouver l'indice d'un élément**
    Écrivez un programme qui demande à l'utilisateur de saisir une liste d'éléments et un élément à rechercher. Affichez l'indice de la première occurrence de cet élément dans la liste.
14. **Concaténer deux listes**
    Écrivez un programme qui demande à l'utilisateur de saisir deux listes d'éléments et affiche la liste résultant de la concaténation des deux listes.
15. **Dupliquer une liste**
    Écrivez un programme qui demande à l'utilisateur de saisir une liste d'éléments et affiche une nouvelle liste qui est une copie de la liste initiale.

16. **Retirer les doublons**
    Écrivez un programme qui demande à l'utilisateur de saisir une liste d'éléments et affiche une nouvelle liste sans doublons.
17. **Trier une liste**
    Écrivez un programme qui demande à l'utilisateur de saisir une liste de nombres et affiche la liste triée dans l'ordre croissant.
18. **Trouver les éléments communs entre deux listes**
    Écrivez un programme qui demande à l'utilisateur de saisir deux listes d'éléments et affiche les éléments communs aux deux listes.
19. **Créer une liste de carrés**
    Écrivez un programme qui demande à l'utilisateur de saisir une liste de nombres et crée une nouvelle liste contenant les carrés de chaque nombre.
20. **Fusionner et trier deux listes**
    Écrivez un programme qui demande à l'utilisateur de saisir deux listes de nombres, fusionne les deux listes, et affiche la liste fusionnée triée.

0.3.2 Niveau 2

1. **Trouver les éléments uniques**
   Écrivez un programme qui demande à l'utilisateur de saisir une liste d'éléments et affiche une nouvelle liste contenant uniquement les éléments uniques (pas de doublons).
2. **Échange des extrémités**
   Écrivez un programme qui demande à l'utilisateur de saisir une liste d'éléments et échange le premier et le dernier élément de la liste. Affichez la liste modifiée.
3. **Liste de paires**
   Écrivez un programme qui prend une liste d'éléments en entrée et génère une liste de paires contenant chaque élément et son indice. Par exemple, pour `['a', 'b', 'c']`, la sortie doit être `[(0, 'a'), (1, 'b'), (2, 'c')]`.
4. **Produits d'une liste**
   Écrivez un programme qui demande à l'utilisateur une liste de nombres et retourne une nouvelle liste où chaque élément est le produit des autres éléments de la liste (sans l'élément courant).
5. **Rassembler les éléments pairs et impairs**
   Écrivez un programme qui demande une liste de nombres à l'utilisateur et retourne deux listes : une contenant les éléments pairs et l'autre les éléments impairs.
6. **Supprimer les valeurs nulles**
   Écrivez un programme qui demande à l'utilisateur de saisir une liste d'éléments (qui peut contenir des valeurs nulles) et affiche une nouvelle liste sans les valeurs nulles (`None`, `0`, `""`, etc.).
7. **Trouver les sous-listes continues dont la somme est zéro**
   Écrivez un programme qui demande une liste de nombres et trouve toutes les sous-listes continues dont la somme des éléments est égale à zéro.
8. **Trouver le sous-ensemble maximal**
   Écrivez un programme qui demande une liste de nombres et retourne la sous-liste avec la somme maximale.
9. **Déplacer les zéros en fin de liste**
   Écrivez un programme qui demande une liste de nombres et déplace tous les zéros en fin de liste, en conservant l'ordre des autres éléments.
10. **Liste de listes**
    Écrivez un programme qui demande à l'utilisateur une liste de nombres et crée une liste de listes, où chaque sous-liste contient les éléments précédents.

Par exemple, pour l'entrée `[1, 2, 3]`, la sortie doit être `[[1], [1, 2], [1, 2, 3]]`.

11. **Vérifier si une liste est palindromique**
    Écrivez un programme qui demande une liste d'éléments et vérifie si elle est identique à l'envers (palindrome).
12. **Compter les éléments pairs et impairs**
    Écrivez un programme qui demande une liste de nombres et affiche le nombre d'éléments pairs et impairs dans la liste.
13. **Rotation circulaire des éléments**
    Écrivez un programme qui demande une liste d'éléments et un nombre entier k. Faites une rotation circulaire de la liste de k positions vers la droite.
14. **Fusion de listes triées**
    Écrivez un programme qui prend en entrée deux listes triées de nombres et les fusionne en une seule liste triée.
15. **Trouver le deuxième plus grand élément**
    Écrivez un programme qui demande une liste de nombres et affiche le deuxième plus grand élément de la liste.
16. **Calcul des différences successives**
    Écrivez un programme qui prend une liste de nombres en entrée et crée une nouvelle liste contenant les différences entre chaque paire d'éléments consécutifs.
17. **Diviser une liste en sous-listes de longueur égale**
    Écrivez un programme qui prend une liste d'éléments et un entier n, et divise la liste en sous-listes de longueur n.
18. **Calcul de la médiane**
    Écrivez un programme qui demande une liste de nombres et affiche la médiane. Assurez-vous de trier la liste et gérer les listes de longueur paire et impaire.
19. **Vérifier les doublons dans une liste**
    Écrivez un programme qui demande une liste d'éléments et indique si la liste contient des doublons.
20. **Trouver les nombres manquants**
    Écrivez un programme qui demande une liste de nombres triée dans un intervalle donné et identifie les nombres manquants de cet intervalle.

0.3.3 Niveau 3

1. **Trouver la sous-liste ayant la plus grande somme**
   Écrivez un programme qui demande une liste de nombres et trouve la sous-liste continue ayant la somme maximale (sous-problème de la somme maximale). Utilisez une approche optimisée, comme l'algorithme de Kadane.
2. **Réorganiser la liste en alternant des valeurs plus grandes et plus petites**
   Écrivez un programme qui réorganise une liste de manière à alterner des valeurs plus grandes et plus petites. Par exemple, transformez `[1, 2, 3, 4, 5, 6]` en `[2, 1, 4, 3, 6, 5]`.
3. **Multiplication cumulée**
   Écrivez un programme qui prend une liste de nombres et crée une nouvelle liste où chaque élément est le produit cumulatif de tous les éléments précédents, sans utiliser de boucles imbriquées. Par exemple, pour la liste `[1, 2, 3, 4]`, la sortie doit être `[1, 1, 2, 6]`.
4. **Trouver tous les triplets qui forment une somme nulle**
   Écrivez un programme qui demande une liste de nombres et trouve tous les triplets uniques de nombres dans la liste dont la somme est égale à zéro. Assurez-vous que les triplets sont uniques, même si les nombres sont répétés dans la liste.

5. **Remplir la liste des valeurs manquantes pour en faire une liste consécutive**
   Écrivez un programme qui prend une liste de nombres non triés et la remplit avec les nombres manquants pour qu'elle devienne une séquence consécutive, de manière croissante, sans changer l'ordre des autres éléments.
6. **Évaluation d'expressions en notation polonaise inverse (RPN)**
   Écrivez un programme qui prend en entrée une expression mathématique en notation polonaise inverse (Reverse Polish Notation) sous forme de liste, puis l'évalue et retourne le résultat. Par exemple, pour l'entrée `[2, 1, "+", 3, "*"]`, la sortie doit être 9.
7. **Liste des carrés sans duplications**
   Écrivez un programme qui prend une liste de nombres et renvoie une nouvelle liste contenant les carrés des nombres de la liste d'origine, sans doublons. Par exemple, pour l'entrée `[1, -1, 2, -2, 3]`, la sortie doit être `[1, 4, 9]`.
8. **Détecter une sous-liste palindrome**
   Écrivez un programme qui prend une liste et détermine si elle contient une sous-liste continue qui est un palindrome. Par exemple, la liste `[3, 5, 6, 7, 6, 5, 3]` contient une sous-liste palindrome.
9. **Équilibrer la somme des sous-listes**
   Écrivez un programme qui prend une liste et trouve un indice où la somme des éléments de gauche est égale à la somme des éléments de droite. Si cet indice n'existe pas, retournez `-1`.
10. **Intercaler deux listes**
    Écrivez un programme qui prend deux listes de longueurs différentes et crée une nouvelle liste qui intercale leurs éléments. Si une des listes est plus longue, placez les éléments restants de cette liste à la fin.

0.4 Exercices sur la récursivité

0.4.1 Niveau 1

1. **Calculer la somme des nombres jusqu'à n**
   Écrivez une fonction récursive `somme(n)` qui retourne la somme des entiers de 1 à n.
2. **Calculer la factorielle de n**
   Écrivez une fonction récursive `factorielle(n)` qui retourne la factorielle de n, définie par $n! = n \times (n-1)!$ et $0! = 1$.
3. **Calculer la puissance d'un nombre**
   Écrivez une fonction récursive `puissance(a, b)` qui retourne a^b (la puissance b de a).
4. **Imprimer une liste d'éléments**
   Écrivez une fonction récursive `imprimer_liste(lst)` qui affiche chaque élément d'une liste `lst`.
5. **Compter les éléments d'une liste**
   Écrivez une fonction récursive `compter_elements(lst)` qui retourne le nombre d'éléments dans une liste `lst`.
6. **Trouver le maximum dans une liste**
   Écrivez une fonction récursive `maximum(lst)` qui retourne le plus grand élément dans une liste `lst`.
7. **Inverser une chaîne de caractères**
   Écrivez une fonction récursive `inverser(chaine)` qui retourne la chaîne `chaine` inversée.

8. **Vérifier si une chaîne est un palindrome**
   Écrivez une fonction récursive `est_palindrome(chaine)` qui vérifie si une chaîne de caractères est un palindrome.
9. **Calculer la somme des chiffres d'un nombre**
   Écrivez une fonction récursive `somme_chiffres(n)` qui retourne la somme des chiffres de n.
10. **Calculer la suite de Fibonacci**
    Écrivez une fonction récursive `fibonacci(n)` qui retourne le n-ième terme de la suite de Fibonacci.
11. **Trouver la longueur d'une chaîne**
    Écrivez une fonction récursive `longueur(chaine)` qui retourne la longueur d'une chaîne `chaine` sans utiliser la fonction `len()`.
12. **Compter les occurrences d'un caractère dans une chaîne**
    Écrivez une fonction récursive `compter_caractere(chaine, c)` qui retourne le nombre de fois que le caractère `c` apparaît dans `chaine`.
13. **Calculer le produit de tous les éléments d'une liste**
    Écrivez une fonction récursive `produit(lst)` qui retourne le produit de tous les éléments d'une liste de nombres `lst`.
14. **Trouver la position d'un élément dans une liste**
    Écrivez une fonction récursive `trouver_position(lst, element)` qui retourne la première position de `element` dans la liste `lst` ou `-1` si l'élément n'est pas trouvé.
15. **Compter les éléments pairs dans une liste**
    Écrivez une fonction récursive `compter_pairs(lst)` qui retourne le nombre d'éléments pairs dans une liste `lst`.
16. **Élever chaque élément d'une liste à une puissance**
    Écrivez une fonction récursive `puissance_liste(lst, p)` qui retourne une nouvelle liste contenant chaque élément de `lst` élevé à la puissance p.
17. **Remplacer toutes les occurrences d'un élément dans une liste**
    Écrivez une fonction récursive `remplacer(lst, ancien, nouveau)` qui remplace toutes les occurrences de `ancien` par `nouveau` dans `lst`.
18. **Fusionner deux chaînes de caractères**
    Écrivez une fonction récursive `fusionner(chaine1, chaine2)` qui retourne une nouvelle chaîne avec les caractères de `chaine1` et `chaine2` fusionnés.
19. **Calculer la somme des nombres d'une liste imbriquée**
    Écrivez une fonction récursive `somme_imbriquee(lst)` qui retourne la somme de tous les nombres dans une liste `lst` qui peut contenir des sous-listes.
20. **Imprimer un nombre en binaire**
    Écrivez une fonction récursive `en_binaire(n)` qui retourne la représentation binaire d'un nombre entier positif n.

0.4.2 Niveau 2

1. **Calculer le nombre de façons d'obtenir une somme avec des dés**
   Écrivez une fonction récursive `combinaisons_des(somme, des)` qui calcule le nombre de façons d'obtenir une somme donnée en lançant un certain nombre de dés à 6 faces.
2. **Lister tous les sous-ensembles d'un ensemble**
   Écrivez une fonction récursive `sous_ensembles(ensemble)` qui retourne tous les sous-ensembles d'un ensemble donné sous forme de liste.
3. **Calculer la puissance de deux en utilisant uniquement la récursivité**
   Écrivez une fonction récursive `puissance_de_deux(n)` qui retourne 2^n en utilisant uniquement des appels récursifs et sans opérateur de puissance.

4. **Vérifier l'équilibre des parenthèses**
   Écrivez une fonction récursive `equilibre_parentheses(chaine)` qui vérifie si une chaîne de caractères contenant des parenthèses est correctement équilibrée.
5. **Trouver la profondeur maximale d'une liste imbriquée**
   Écrivez une fonction récursive `profondeur_maximale(lst)` qui retourne la profondeur maximale d'une liste contenant des sous-listes imbriquées.
6. **Calculer le produit scalaire de deux vecteurs**
   Écrivez une fonction récursive `produit_scalaire(vecteur1, vecteur2)` qui calcule le produit scalaire de deux listes de nombres de même longueur.
7. **Compter le nombre d'occurrences d'un mot dans une chaîne**
   Écrivez une fonction récursive `compter_mots(chaine, mot)` qui compte le nombre d'occurrences d'un mot donné dans une chaîne de caractères.
8. **Vérifier si une liste est triée**
   Écrivez une fonction récursive `est_triee(lst)` qui retourne `True` si la liste `lst` est triée en ordre croissant, sinon retourne `False`.
9. **Trouver tous les chemins dans une grille**
   Écrivez une fonction récursive `nombre_de_chemins(m, n)` qui retourne le nombre de chemins possibles d'une position $(0, 0)$ à (m, n) dans une grille, en ne se déplaçant que vers le bas ou la droite.
10. **Calculer la somme des éléments de la diagonale principale d'une matrice**
    Écrivez une fonction récursive `somme_diagonale(matrice)` qui retourne la somme des éléments de la diagonale principale d'une matrice carrée.
11. **Diviser une liste en deux parties égales**
    Écrivez une fonction récursive `diviser_liste(lst)` qui divise une liste en deux sous-listes d'éléments aussi proches que possible en taille.
12. **Trouver le nombre de façons de monter un escalier**
    Écrivez une fonction récursive `façons_escalier(n)` qui calcule le nombre de façons de monter un escalier de n marches, sachant que l'on peut monter 1, 2 ou 3 marches à chaque pas.
13. **Générer toutes les permutations d'une chaîne de caractères**
    Écrivez une fonction récursive `permutations(chaine)` qui génère toutes les permutations possibles d'une chaîne de caractères donnée.
14. **Calculer la somme des éléments d'une liste imbriquée, en doublant la profondeur**
    Écrivez une fonction récursive `somme_profondeur(lst)` qui calcule la somme de tous les éléments d'une liste imbriquée, en multipliant chaque élément par la profondeur de la liste où il se trouve.
15. **Vérifier si deux chaînes sont des anagrammes**
    Écrivez une fonction récursive `est_anagramme(chaine1, chaine2)` qui vérifie si deux chaînes sont des anagrammes.
16. **Inverser une pile en utilisant la récursivité**
    Écrivez une fonction récursive `inverser_pile(pile)` qui prend une pile représentée sous forme de liste et la renverse en utilisant uniquement des appels récursifs.
17. **Trouver le plus grand nombre dans une liste de listes imbriquées**
    Écrivez une fonction récursive `maximum_imbrique(lst)` qui retourne le plus grand nombre dans une liste contenant des sous-listes imbriquées.
18. **Compresser une chaîne de caractères en supprimant les doublons consécutifs**
    Écrivez une fonction récursive `compresser(chaine)` qui retourne une nouvelle chaîne en supprimant les doublons consécutifs de la chaîne d'entrée.
19. **Vérifier si une liste contient une séquence cible**
    Écrivez une fonction récursive `contient_sequence(lst, sequence)` qui vérifie si une liste `lst` contient une séquence spécifique de valeurs.

20. **Fusionner deux listes triées en une seule liste triée**
    Écrivez une fonction récursive `fusion_listes(lst1, lst2)` qui fusionne deux listes triées en une seule liste triée.

```python
# Exercice 1 : Calculer le nombre de façons d'obtenir une somme avec des dés
def combinaisons_des(somme, des):
    if somme < 0:
        return 0
    if somme == 0:
        return 1
    if des == 0:
        return 0
    return combinaisons_des(somme - 1, des - 1) + combinaisons_des(somme - 6, des - 1)

# Exercice 2 : Lister tous les sous-ensembles d'un ensemble
def sous_ensembles(ensemble):
    if not ensemble:
        return [[]]
    sous_ens = sous_ensembles(ensemble[1:])
    return sous_ens + [[ensemble[0]] + sous for sous in sous_ens]

# Exercice 3 : Calculer la puissance de deux en utilisant uniquement la récursivité
def puissance_de_deux(n):
    if n == 0:
        return 1
    return 2 * puissance_de_deux(n - 1)

# Exercice 4 : Vérifier l'équilibre des parenthèses
def equilibre_parentheses(chaine, count=0):
    if count < 0:
        return False
    if not chaine:
        return count == 0
    if chaine[0] == '(':
        return equilibre_parentheses(chaine[1:], count + 1)
    elif chaine[0] == ')':
        return equilibre_parentheses(chaine[1:], count - 1)
    return equilibre_parentheses(chaine[1:], count)

# Exercice 5 : Trouver la profondeur maximale d'une liste imbriquée
def profondeur_maximale(lst):
    if not isinstance(lst, list):
        return 0
    return 1 + max(profondeur_maximale(sous_lst) for sous_lst in lst)

# Exercice 6 : Calculer le produit scalaire de deux vecteurs
def produit_scalaire(vecteur1, vecteur2):
    if not vecteur1 or not vecteur2:
        return 0
    return vecteur1[0] * vecteur2[0] + produit_scalaire(vecteur1[1:], vecteur2[1:])

# Exercice 7 : Compter le nombre d occurrences d'un mot dans une chaîne
def compter_mots(chaine, mot):
    if len(chaine) < len(mot):
        return 0
```

```python
    if chaine[:len(mot)] == mot:
        return 1 + compter_mots(chaine[len(mot):], mot)
    return compter_mots(chaine[1:], mot)

# Exercice 8 : Vérifier si une liste est triée
def est_triee(lst):
    if len(lst) <= 1:
        return True
    return lst[0] <= lst[1] and est_triee(lst[1:])

# Exercice 9 : Trouver tous les chemins dans une grille
def nombre_de_chemins(m, n):
    if m == 0 or n == 0:
        return 1
    return nombre_de_chemins(m - 1, n) + nombre_de_chemins(m, n - 1)

# Exercice 10 : Calculer la somme des éléments de la diagonale principale d'
    une matrice
def somme_diagonale(matrice, i=0):
    if i == len(matrice):
        return 0
    return matrice[i][i] + somme_diagonale(matrice, i + 1)

# Exercice 11 : Diviser une liste en deux parties égales
def diviser_liste(lst):
    if not lst:
        return [], []
    partie1, partie2 = diviser_liste(lst[2:])
    return [lst[0]] + partie1, (partie2 if len(lst) < 2 else [lst[1]] +
    partie2)

# Exercice 12 : Trouver le nombre de façons de monter un escalier
def facons_escalier(n):
    if n == 0:
        return 1
    if n < 0:
        return 0
    return facons_escalier(n - 1) + facons_escalier(n - 2) + facons_escalier(n
     - 3)

# Exercice 13 : Générer toutes les permutations d'une chaîne de caractères
def permutations(chaine):
    if len(chaine) == 0:
        return [""]
    perm = []
    for i in range(len(chaine)):
        for p in permutations(chaine[:i] + chaine[i+1:]):
            perm.append(chaine[i] + p)
    return perm

# Exercice 14 : Calculer la somme des éléments d'une liste imbriquée, en
    doublant la profondeur
def somme_profondeur(lst, profondeur=1):
    total = 0
    for el in lst:
        if isinstance(el, list):
            total += somme_profondeur(el, profondeur + 1)
        else:
```

```
            total += el * profondeur
    return total

# Exercice 15 : Vérifier si deux chaînes sont des anagrammes
def est_anagramme(chaine1, chaine2):
    if sorted(chaine1) == sorted(chaine2):
        return True
    return False

# Exercice 16 : Inverser une pile en utilisant la récursivité
def inverser_pile(pile):
    if not pile:
        return []
    val = pile.pop()
    res = inverser_pile(pile)
    res.insert(0, val)
    return res

# Exercice 17 : Trouver le plus grand nombre dans une liste de listes imbriqué
    es
def maximum_imbrique(lst):
    max_val = float('-inf')
    for el in lst:
        if isinstance(el, list):
            max_val = max(max_val, maximum_imbrique(el))
        else:
            max_val = max(max_val, el)
    return max_val

# Exercice 18 : Compresser une chaîne de caractères en supprimant les doublons
     consécutifs
def compresser(chaine):
    if not chaine:
        return ""
    if len(chaine) == 1:
        return chaine
    if chaine[0] == chaine[1]:
        return compresser(chaine[1:])
    return chaine[0] + compresser(chaine[1:])

# Exercice 19 : Vérifier si une liste contient une séquence cible
def contient_sequence(lst, sequence):
    if not sequence:
        return True
    if not lst:
        return False
    if lst[:len(sequence)] == sequence:
        return True
    return contient_sequence(lst[1:], sequence)

# Exercice 20 : Fusionner deux listes triées en une seule liste triée
def fusion_listes(lst1, lst2):
    if not lst1:
        return lst2
    if not lst2:
        return lst1
    if lst1[0] < lst2[0]:
        return [lst1[0]] + fusion_listes(lst1[1:], lst2)
```

```
    else:
        return [lst2[0]] + fusion_listes(lst1, lst2[1:])
```

0.4.3 Niveau 3

1. **Tuilage de Domino d'un Plateau $2 \times n$**
   Écrivez une fonction récursive `tuilage(n)` qui calcule le nombre de façons possibles de recouvrir un plateau de $2 \times n$ avec des dominos 1×2.
2. **Tour de Hanoï optimisée avec n disques**
   Écrivez une fonction récursive `tour_de_hanoi(n, depart, destination, auxiliaire)` qui affiche les étapes nécessaires pour résoudre la tour de Hanoï avec n disques et trois poteaux. Comptez le nombre total de déplacements nécessaires.
3. **Calculer la somme des chemins d'une matrice avec des poids**
   Écrivez une fonction récursive `somme_chemins(matrice, x, y)` qui retourne la somme maximale possible en partant de la position $(0, 0)$ et en arrivant en bas à droite (x, y) de la matrice. On ne peut se déplacer que vers le bas ou vers la droite.
4. **Diviser une chaîne en sous-chaînes palindromiques**
   Écrivez une fonction récursive `partition_palindromes(chaine)` qui divise une chaîne de caractères en sous-chaînes de manière à ce que chaque sous-chaîne soit un palindrome. Affichez toutes les partitions possibles.
5. **Calculer les arrangements d'objets avec des répétitions limitées**
   Écrivez une fonction récursive `arrangements(ensemble, limite)` qui calcule toutes les dispositions possibles d'un ensemble d'éléments où chaque élément peut être utilisé jusqu'à un certain nombre de fois défini par `limite`.
6. **Compresser une liste de chaînes par fréquence**
   Écrivez une fonction récursive `compresser_frequence(lst)` qui compresse une liste de chaînes de caractères en indiquant la fréquence de chaque chaîne. Par exemple, `["a", "a", "b", "b", "b", "c"]` devient `["a:2", "b:3", "c:1"]`.
7. **Résoudre le problème de la somme de sous-ensemble**
   Écrivez une fonction récursive `somme_sous_ensemble(lst, cible)` qui détermine s'il existe un sous-ensemble dans une liste de nombres `lst` qui a pour somme un nombre donné `cible`. Retournez `True` si un tel sous-ensemble existe, sinon `False`.
8. **Calculer le produit cartésien de plusieurs listes**
   Écrivez une fonction récursive `produit_cartesien(liste_de_listes)` qui retourne le produit cartésien de plusieurs listes. Par exemple, pour `[[1, 2], [3, 4]]`, la sortie doit être `[(1,3), (1,4), (2,3), (2,4)]`.
9. **Trouver tous les chemins uniques dans un labyrinthe**
   Écrivez une fonction récursive `chemins_labyrinthe(labyrinthe, x, y, chemin)` qui trouve tous les chemins uniques possibles dans un labyrinthe représenté par une matrice. Les cases valides sont indiquées par 1 et les obstacles par 0.
10. **Résoudre un Sudoku partiellement rempli**
    Écrivez une fonction récursive `resoudre_sudoku(grille)` qui résout un Sudoku partiellement rempli, représenté par une grille 9×9. Utilisez une approche de backtracking pour remplir chaque case vide avec les nombres de 1 à 9.

```
# Exercice 1 : Tuilage de Domino d'un Plateau 2 x n
def tuilage(n):
    if n == 0 or n == 1:
        return 1
    return tuilage(n - 1) + tuilage(n - 2)

```

```python
# Exercice 2 : Tour de HanoÃ¯ optimisée avec n disques
def tour_de_hanoi(n, depart, destination, auxiliaire):
    if n == 1:
        print(f"Déplace le disque 1 de {depart} vers {destination}")
        return 1
    moves = tour_de_hanoi(n - 1, depart, auxiliaire, destination)
    print(f"Déplace le disque {n} de {depart} vers {destination}")
    moves += 1
    moves += tour_de_hanoi(n - 1, auxiliaire, destination, depart)
    return moves

# Exercice 3 : Calculer la somme des chemins d'une matrice avec des poids
def somme_chemins(matrice, x, y):
    if x == 0 and y == 0:
        return matrice[0][0]
    elif x < 0 or y < 0:
        return float('-inf')
    else:
        return matrice[x][y] + max(somme_chemins(matrice, x-1, y),
    somme_chemins(matrice, x, y-1))

# Exercice 4 : Diviser une chaîne en sous-chaînes palindromiques
def est_palindrome(s):
    return s == s[::-1]

def partition_palindromes(chaine, resultat=None):
    if resultat is None:
        resultat = []
    if not chaine:
        print(resultat)
        return
    for i in range(1, len(chaine) + 1):
        prefix = chaine[:i]
        if est_palindrome(prefix):
            partition_palindromes(chaine[i:], resultat + [prefix])

# Exercice 5 : Calculer les arrangements d'objets avec des répétitions limité
    es
def arrangements(ensemble, limite, courant=[]):
    if sum(courant) == limite:
        print(courant)
        return
    for i in ensemble:
        if sum(courant) + i <= limite:
            arrangements(ensemble, limite, courant + [i])

# Exercice 6 : Compresser une liste de chaînes par fréquence
def compresser_frequence(lst, index=0, resultat=None):
    if resultat is None:
        resultat = []
    if index >= len(lst):
        return resultat
    count = 1
    while index + count < len(lst) and lst[index] == lst[index + count]:
        count += 1
    resultat.append(f"{lst[index]}:{count}")
    return compresser_frequence(lst, index + count, resultat)

```

```python
# Exercice 7 : Résoudre le problème de la somme de sous-ensemble
def somme_sous_ensemble(lst, cible, index=0):
    if cible == 0:
        return True
    if index >= len(lst) or cible < 0:
        return False
    return somme_sous_ensemble(lst, cible - lst[index], index + 1) or somme_sous_ensemble(lst, cible, index + 1)

# Exercice 8 : Calculer le produit cartésien de plusieurs listes
def produit_cartesien(liste_de_listes, index=0, courant=[]):
    if index == len(liste_de_listes):
        print(tuple(courant))
        return
    for element in liste_de_listes[index]:
        produit_cartesien(liste_de_listes, index + 1, courant + [element])

# Exercice 9 : Trouver tous les chemins uniques dans un labyrinthe
def chemins_labyrinthe(labyrinthe, x, y, chemin=[]):
    if x >= len(labyrinthe) or y >= len(labyrinthe[0]) or labyrinthe[x][y] == 0:
        return
    if (x, y) == (len(labyrinthe) - 1, len(labyrinthe[0]) - 1):
        print(chemin + [(x, y)])
        return
    labyrinthe[x][y] = 0  # marquer comme visité
    chemins_labyrinthe(labyrinthe, x + 1, y, chemin + [(x, y)])
    chemins_labyrinthe(labyrinthe, x, y + 1, chemin + [(x, y)])
    labyrinthe[x][y] = 1  # annuler la visite

# Exercice 10 : Résoudre un Sudoku partiellement rempli
def est_valide(grille, ligne, col, num):
    for i in range(9):
        if grille[ligne][i] == num or grille[i][col] == num:
            return False
    bloc_x, bloc_y = 3 * (ligne // 3), 3 * (col // 3)
    for i in range(3):
        for j in range(3):
            if grille[bloc_x + i][bloc_y + j] == num:
                return False
    return True

def resoudre_sudoku(grille):
    for i in range(9):
        for j in range(9):
            if grille[i][j] == 0:
                for num in range(1, 10):
                    if est_valide(grille, i, j, num):
                        grille[i][j] = num
                        if resoudre_sudoku(grille):
                            return True
                        grille[i][j] = 0
                return False
    return True
```