new_board

```python
def new_board(n):  1 usage
    return [[0 for _ in range(n)] for _ in range(n)]
```

Permet de générer le tableau de n lignes avec n colonnes.

Je n'ai pas pu utiliser [[0] * n] * n sinon les colonnes auraient été liées entre elles.

display_board

```python
def display_board(board, n):  2 usages
    max_length = 0
    for i in range(n):
        text = str(i + 1) + (len(str(n)) - len(str(i + 1))) * " " + " | "
        for j in range(n):
            text += ("." if board[i][j] == 0 else ("x" if board[i][j] == 1 else "o")) + " " * len(str(n))
        print(text)
        if max_length < len(text):
            max_length = len(text)

    print(" " * (len(str(n)) + 1), end="")
    for i in range(max_length - 3):
        print("-", end="")
    print()

    print(len(str(n)) * " " + 3 * " ", end="")

    for i in range(n):
        print(i + 1, end=" " + " " * (len(str(n)) - len(str(i + 1))))
    print()
```

Fonction qui affiche le tableau correctement sur un nombre infini de case (si les performances de l'ordinateur le permettent).

Chaque ligne est créé dans une variable avec son début (numéro de ligne, | et les espaces) puis pour chaque élément, je le rajoute à la variable (espaces + . ou o ou x). Les espaces sont calculés grâce à la longueur du nombre à afficher en chaine de caractère.

Je regarde ensuite la longueur du texte pour savoir la longueur maximale afin de mettre la bonne quantité de tirets pour la ligne dessous.

Enfin, je reprends le même principe que pour afficher les numéros de lignes mais je l'utilise pour les colonnes. Pour les espaces, j'utilise la différence de longueur des dimensions du tableau en string afin d'avoir le bon nombre d'espace en fonction du nombre à afficher.

possible_square

```
def possible_square(board, n, player, i, j):  2 usages
    if i >= n or i < 0 or j >= n or j < 0:
        return False
    other_player = get_other_player(player)
    if n > i - 1 >= 0 and board[i - 1][j] == other_player:
        return False
    if n > i + 1 >= 0 and board[i + 1][j] == other_player:
        return False
    if n > j - 1 >= 0 and board[i][j - 1] == other_player:
        return False
    if n > j + 1 >= 0 and board[i][j + 1] == other_player:
        return False
    return board[i][j] == 0
```

Cette fonction permet de vérifier si les coordonnées sont prenables.

Dans le premier if, je vérifie si la case voulu est dans le tableau, en l'occurrence dans ce code je fonctionne à l'inverse, j'élimine les intrus dès le début afin de garder un code lisible sans trop d'indentations.

Vu que l'on souhaite vérifier que la case en à gauche, droite, en bas et en haut soit dans le tableau, que la case soit vide, je fonctionne de manière inversée dans la vérification du contenu de la case, car, si c'est le joueur lui-même qui est

présent, il peut quand même jouer, ou alors si la case est vide donc j'utilise la valeur de l'autre joueur qui elle seule est bloquante.

Je peux donc réduire l'if en créant une variable qui possède la valeur opposée du joueur de vérifier cela dans la case, dans ce cas-là, je renverrais False.

Pour finir je renvoi une dernière condition, celle-ci non inversé par rapport aux autres, elle permet de vérifier si la case visée est effectivement vide.

Certes, je pourrai optimiser cela en la mettant en deuxième voir en première position dans cette fonction, mais pour plus de compréhension j'ai souhaité la laisser ici.

select_square

```
def select_square(board, n, player):  2 usages
    return int(input("Choisir un numéro de ligne : ")) - 1, int(input("Choisir un numéro de colonne : ")) - 1
```

Petite fonction pour demander au joueur la ligne et la colonne voulue, on enlève 1 directement car on lui a demandé un nombre entre 1 et 4 alors que dans le code on fonctionne entre 0 et 3.

update_board

```
def update_board(board, player, i, j):  1 usa
    board[i][j] = player
```

Permet de mettre à jour la case avec la valeur du joueur (ici soit 1 ou 2)

get_other_player

```python
def get_other_player(player):  3 usages
    return 1 if player == 2 else 2
```

Fonction qui n'était pas demandée mais qui me permet de garder un code propre. Elle permet d'inverser le joueur.

again

```python
def again(board, n, player):  2 usages
    for i in range(n):
        for j in range(n):
            if possible_square(board, n, player, i, j):
                return True
    return False
```

Fonction qui permet de vérifier si le joueur peut encore jouer. Je réutilise la fonction possible square car la logique est la même.

Certes je pourrai aussi vérifier si la case est vide avant de lancer la fonction mais je le vérifie déjà dans celle-ci.

snort

```python
def snort(n):  1 usage
    if n <= 0:
        print("Merci de chosir un plateau positif")
        return
    board = new_board(n)
    display_board(board, n)
    current_player = 1

    while again(board, n, current_player):
        print("Au joueur", current_player, "de jouer")

        i, j = select_square(board, n, current_player)
        while not possible_square(board, n, current_player, i, j):
            i, j = select_square(board, n, current_player)

        update_board(board, current_player, i, j)
        display_board(board, n)

        current_player = get_other_player(current_player)

    current_player = get_other_player(current_player)
    print("Le gagnant est le joueur", current_player)
```

La fonction snort permet de lancer une partie et de la gérer en entier.

Au début, j'initialise une variable avec le tableau de jeux. Si le tableau est trop petit, on envoie un message d'erreur.

J'affiche le plateau au démarrage afin que les joueurs prennent connaissance du plateau.

J'initialise aussi une variable nommée current_player qui évolura au cours du temps et des tours grâce à la fonction pour inverser le joueur.

Ensuite, j'utilise une boucle while avec la fonction again afin de faire tourner le jeu tant que le joueur peut jouer. Sachant que le joueur échangera entre 1 et 2 au fil du temps et des tours.

Ensuite dans la boucle while, je demande à l'utilisateur 1 ou 2 en fonction du tour, quelle coordonnée il souhaite prendre, tant que celles-ci ne sont pas prenables, il devra les re rentrer.

Une fois la coordonnée choisie, je mets à jour le tableau avec la fonction update puis je le réaffiche une fois modifié.

Juste après, j'inverse le joueur.

Si le joueur une fois inversé ne peut plus jouer, la boucle se termine, il faut donc le re inverser pour annoncer le bon gagnant.

Lancer le jeu

```
snort(3)
```

Lance la fonction avec une taille donnée en argument ici 3.

A noter qu'il faut une dimension de 1 minimum pour que le jeu soit jouable, une condition dans la fonction snort empêche de démarrer le jeu si le plateau est trop petit.