CORRIGÉ DU SUJET 2026

Partie I. Des algorithmes pour résoudre le problème des n reines

I.1. Introduction

Question 1. La seconde configuration ne résout pas le problème car il y a deux reines sur la même diagonale (en (1,2) et (2,3)).

Question 2. `L1 = [2, 0, 3, 1]` pour la première configuration et `L2 = [0, 2, 3, 1]` pour la seconde.

I.2. Mise en place d'un affichage

Question 3.

```
def ligne(n):
    return [0]*n
```

ou bien

```
def ligne(n):
    return [0 for k in range(n)]
```

D'autres réponses sont possibles (avec des append successifs par exemple)

Question 4.

```
def grille(n):
    G = []
    for i in range(n):
        G.append(ligne(n))
    return G
```

Question 5.

```
def remplir(L):
    n = len(L)
    G = grille(n)
    for i in range(n):
        G[i][L[i]] = 1
    return G # en fait modification en place de la grille
```

Question 6.

```
def affiche(G):
    for ligne in G:
        print(ligne)
```

I.3. Tester si une configuration est valide

Question 7. La condition (a) assure qu'il n'y a pas deux reines sur la même colonne. Les conditions (b) et (c) assurent qu'il n'y a pas deux reines sur la même diagonale.

Ce qui précède est suffisant. En fait pour être plus précis, on peut remarquer que les diagonales montantes ont pour équation $i+j=k$ (d_k) avec $k \in [\![1, n+1]\!]$ et les diagonales descendantes ont pour équation $i-j=n-2-k$ (δ_k) avec $k \in [\![0, 2n-4]\!]$. Donc, par exemple, `(i, L[i])` *et* `(i', L[i'])` *sont sur (d_k) si, et seulement si,* `i+L[i] = k = i'+L[i']`*...*

Question 8.

```
def tous_distincts(L):
    """version en O(n) avec dictionnaire"""
    presents = dict()
    for elt in L:
        if elt not in presents:
            presents[elt] = True
        else:
            return False
    return True
```

Question 9. Voici une version corrigée :

```
def tous_distincts2(L):
    """version  avec liste en O(n^2)"""
    n = len(L)
    for i in range(n):
        for j in range(i+1, n):
            if L[i]  == L[j]:
                return False
    return True
```

La complexité est en $O(n^2)$ (deux boucles imbriquées). La version de la question 8 est en $O(n)$ (un seul parcours de la liste). Le code de la question 8 est donc asymptotiquement beaucoup plus rapide.

Question 10.

```
def est_possible(L) :
    n = len(L)
    test1 = tous_distincts(L)
    L2 = [L[j] + j for j in range(n)]
    test2 = tous_distincts(L2)
    L3 = [L[j] - j for j in range(n)]
    test3 = tous_distincts(L3)
    return test1 and test2 and test3
```

Question 11.

```
def est_solution(L, n) :
    return est_possible(L) and len(L) == n
```

I.4. Une première approche par recherche aléatoire

Question 12.

```
L = [1, 2, 3]
from random import sample # ou from random import *
L1 = sample(L, len(L))
```

```
L = [1, 2, 3]
import random # ou, à la limite, import random as random
L1 = random.sample(L, len(L))
```

```
L = [1, 2, 3]
import random as rd
L1 = rd.sample(L, len(L))
```

Question 13.

```
def recherche_aleatoire(n, N):
    L = [k  for k in range(n)]
    for k in range(N):
        L1 = sample(L,n)
        if est_solution(L1, n):
            return L1
    return []
```

I.5. Déterminer toutes les permutations de $[\![0, n-1]\!]$

Question 14.

```
def insertion(x, L, i):
    return L[:i] + [x] + L[i:]
```

ou, par exemple,

```
def insertion1(x, L, i):
    L1 = []
    for k in range(i):
        L1.append(L[k])
    L1.append(x)
    for k in range(i, len(L)):
        L1.append(L[k])
    return L1
```

Question 15. Au bout d'un passage dans la première boucle, on a
`P = [[2, 0, 1], [0, 2, 1], [0, 1, 2]]`.
À l'issue de l'exécution complète du code, on a
`P = [[2, 0, 1], [0, 2, 1], [0, 1, 2], [2, 1, 0], [1, 2, 0], [1, 0, 2]]`.
Le code permet de construire toutes les permutations de $[\![0,2]\!]$ en insérant 2 à toutes les positions possibles dans chacune des listes composant `L`.

Question 16.

```
def permutations(n) :
    if n == 0 :
        return [[]]
    L = [[0]]
    for v in range(1, n) :
        P = []
        for l1 in L :
            for i in range(v+1) :
                p1 = insertion(v, l1, i)
                P.append(p1)
        L = P
    return L
```

Question 17.

```
def configurations(n) :
    return [c for c in permutations(n) if est_solution(c, n)]
```

ou, par exemple,


```
def configurations2(n) :
    P = permutations(n)
    C = []
    for p in P:
        if est_solution(p, n):
            C.append(p)
    return C
```

I.6. Classer les configurations similaires

Question 18. `L = [2, 4, 1, 3, 0]`, `L1 = [0, 3, 1, 4, 2]`, `L2 = [0, 2, 4, 1, 3]`
On obtient la liste `L1` en renversant la liste `L`.

Question 19.

```
def symetrie(L):
    n = len(L)
    newL = [0]*n
    for i in range(n):
        newL[i] = L[n-1-i]
    return newL
```

Question 20.

```
def rotation(L):
    n = len(L)
    L_rot = n*[0]
    for i in range(n):
        L_rot[L[i]] = n-1-i
    return L_rot
```

Question 21. Le code qui suit est juste donné à titre indicatif. Les réponses aux questions de l'énoncé suivent.

```
def similaires(L1, L2):
        """ Bonus, ce n' est pas demandé!"""
    L1b = symetrie(L1)
    if L2 == L1 or L2 == L1b:
        return True
    if L2 == rotation(L1) or L2 == rotation(L1b):
        return True
    if  L2 == rotation(rotation(L1)) or L2 == rotation(rotation(L1b)):
        return True
    if L2 == rotation(rotation(rotation(L1))) or L2 == rotation(rotation(rotation(L1b))\
     ):
        return True
    return False

def partition(n):
    C = configurations(n)
    P = []
    for c in C:
        test = False
        for elt in P:
            if similaires(elt[0], c):
                elt.append(c)
                test = True
        if test == False:
            P.append([c])
    return P
```

Que fait la ligne 1 de la fonction `partition` *?*
Elle crée la liste `C` de toutes les configurations gagnantes pour une grille de taille `n×n`.

Quel est le type de la variable renvoyée par l'appel `partition(n)` *?*
Une liste.

Décrire complètement la structure de cette variable.
Une liste de listes de listes ou une liste de listes de configurations.

Expliquer en quelques phrases comment fonctionne cette fonction `partition`.
Pour chaque configuration gagnante `c`, on regarde si une configuration similaire est déjà dans une sous-liste de `P`. Si c'est le cas, on ajoute `c` à cette sous-liste. Sinon on crée une nouvelle sous-liste de `P` contenant cette configuration `c`.

I.7. Des parcours de graphes pour aller plus vite et plus loin

Question 22.

```
def explore(L, n):
        if not est_possible(L):
            return -1
        if len(L) == n:
            return L
        for i in range(n):
            res = explore(L + [i], n)
            if res != -1:
                return res
        return -1

def resout_graphe(n):
    rep = explore([], n)
    return rep
```

Question 23.

```
def explore2(L, n):
        if not est_possible(L):
            return 0
        if len(L) == n:
            return 1
        nb_sol = 0
        for i in range(n):
            nb_sol = nb_sol + explore2(L + [i], n)
        return nb_sol

def compte(n):
    rep = explore2([], n)
    return rep
```

Partie II - Base de données d'un serveur de jeux en ligne

Question 24.

```
SELECT nom, prenom, email
FROM Clients
WHERE anniversaire = '03-07'
```


Question 25.

```
SELECT nom, prenom, date_achat
FROM Clients c JOIN Factures f
ON c.id_client = f.id_client
WHERE email = 'myriam.sarr@ccinp.edu'
```

Question 26.

```
SELECT nom, email, SUM(montant)
FROM Clients c JOIN Factures f
ON c.id_client = f.id_client
GROUP BY id_client
```

Question 27.

```
SELECT j.nom_jeu,  MAX(r.score)
FROM Records  r JOIN Jeux  j
ON r.id_jeu = j.id_jeu
GROUP BY id_jeu
```