MINISTÈRE
DE L'ÉDUCATION
NATIONALE,
DE LA JEUNESSE
ET DES SPORTS
Liberté
Égalité
Fraternité

EAE INF 3

SESSION 2022

AGREGATION
CONCOURS EXTERNE

Section : INFORMATIQUE

ÉPREUVE SPÉCIFIQUE SELON L'OPTION CHOISIE :
- **ÉTUDE DE CAS INFORMATIQUE**
- **FONDEMENTS DE L'INFORMATIQUE**

Durée : 6 heures

L'usage de tout ouvrage de référence, de tout dictionnaire et de tout matériel électronique (y compris la calculatrice) est rigoureusement interdit.

Si vous repérez ce qui vous semble être une erreur d'énoncé, vous devez le signaler très lisiblement sur votre copie, en proposer la correction et poursuivre l'épreuve en conséquence. De même, si cela vous conduit à formuler une ou plusieurs hypothèses, vous devez la (ou les) mentionner explicitement.

NB : Conformément au principe d'anonymat, votre copie ne doit comporter aucun signe distinctif, tel que nom, signature, origine, etc. Si le travail qui vous est demandé consiste notamment en la rédaction d'un projet ou d'une note, vous devrez impérativement vous abstenir de la signer ou de l'identifier.

Tournez la page S.V.P.

Épreuve spécifique

INFORMATION AUX CANDIDATS

Vous trouverez ci-après les codes nécessaires vous permettant de compléter les rubriques figurant en en-tête de votre copie.

Ces codes doivent être reportés sur chacune des copies que vous remettrez.

► **Etude de cas informatique** :

Concours	Section/option	Epreuve	Matière
EAE	6200A	103	9424

► **Fondement de l'informatique** :

Concours	Section/option	Epreuve	Matière
EAE	6200A	103	9425

Étude de cas informatique

Préliminaires

Le projet proposé s'inspire d'un projet réel. La procédure de conception utilisée dans le sujet s'inspire librement de l'approche agile avec une organisation en plusieurs parties. Chaque partie comprend la définition d'un certain nombre d'objectifs concrets, la réflexion sur les moyens de les atteindre, la préparation d'une procédure de validation permettant de vérifier si les objectifs sont atteints et une discussion critique de l'ensemble du processus.

Attendus. Il est attendu des candidates et des candidats des réponses construites. Ils seront aussi évalués sur la précision, le soin et la clarté de la rédaction.

Dépendances. Ce sujet contient quatre parties. La première partie est guidée de manière assez précise car elle sert de point de départ à l'ensemble des autres parties. Les autres parties sont largement indépendantes. Il est possible de les aborder dans l'ordre qui vous conviendra le mieux.

Partie I. Problème d'échecs

Le but de ce sujet est de développer la plate-forme **ChessMate** permettant de spécifier et résoudre des problèmes d'échecs tels que le problème des huit dames mais aussi d'autres problèmes tels que les problèmes des mats en N coups. **ChessMate** permettra également de pouvoir jouer une partie d'échecs.

Fonction de résolution du problème des huit dames

Nous allons commencer par le problème des huit dames : "Le but du **problème des huit dames** est de placer huit dames d'un jeu d'échecs sur un échiquier de 8 x 8 cases sans que les dames ne puissent se menacer mutuellement, conformément aux règles du jeu d'échecs (la couleur des pièces étant ignorée). Par conséquent, deux dames ne devraient jamais partager la même rangée, colonne, ou diagonale." (*Problème des huit dames, d'après Wikipedia*)

Le problème des huit dames a 92 solutions distinctes. La figure 1 en présente une. On propose la fonction *check_solution8queens* en Python pour vérifier si une solution au problème des huit dames est correcte. Cette fonction trie les cases où sont positionnées les dames sur l'échiquier et vérifie que les dames ne sont pas sur la même rangée ou colonne. Du moins c'est l'intention du programmeur.

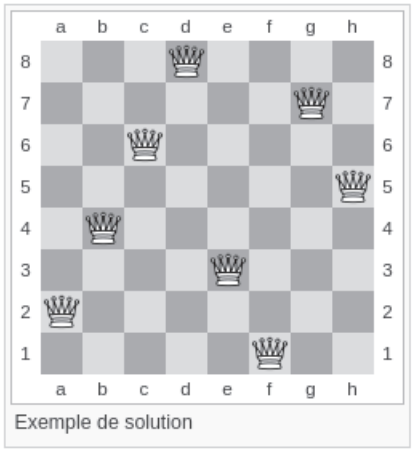

FIGURE 1 – Solution du problème des huit dames

Listing 1 – fonction check_solution8queens

```
1
2  from math import floor
3
4  def check_solution8queens(sol):
5      """
6      sol est une solution au probl\'eme des 8 dames
7      sol est un ensemble de 8 valeurs
8      exemple: [8, 5, 20, 25, 42, 59, 54, 39]
9      correspondant \'a une solution correcte (cf \'echiquier ci-dessus)
10     retourne True si la solution est correcte, False sinon
11
12     l'\'echiquier est repr\'esent\'e par une liste de 64 cases
13     la case 0 est le coin bas gauche
14     la case 63 est le coin haut droit
15     56 57 58 59 60 61 62 63
16     48 49 50 51 52 53 54 55
17     40 41 42 43 44 45 46 47
18     32 33 34 35 36 37 38 39
19     24 25 26 27 28 29 30 31
20     16 17 18 19 20 21 22 23
21     8  9  10 11 12 13 14 15
22     0  1  2  3  4  5  6  7
23     """
24
25     sortedSol = sorted(sol)
26     for i in range(len(sortedSol)):
27         square = sortedSol[i]
28         otherSquares = sortedSol[i+1:]
29         for otherSquare in otherSquares:
30             if square % 8 == otherSquare % 8:
31                 return False
32             if (floor(otherSquare/8) - floor(square/8)) == 0:
33                 return False
34
35     return True
```

Question 1. La fonction du listing 1 est en fait fausse car incomplète. Elle retourne True pour certaines solutions qui ne sont en fait pas correctes. Programmer trois tests qui montrent trois défauts différents de la fonction *check_ solution8queens(sol)*. Le code de vos tests doit préciser

les données en entrée et le résultat attendu. Commenter en quoi vos trois tests vérifient des propriétés différentes.

L'échiquier avec une classe

Maintenant que nous avons testé la fonction *check_solution8queens(sol)*, on s'intéresse à l'encodage de l'échiquier. L'encodage proposé dans la fonction *check_solution8queens(sol)* n'est pas idéal. En principe, une case est identifiée par une lettre pour la colonne (de 'a' à 'h') et un chiffre pour la ligne(de 1 à 8). Ainsi la case 'a1' correspond à la case 0 de l'encodage proposé à la question 1, la case 'b1' correspond à la case 1, etc.

Pour réaliser cet encodage, nous proposons la classe **Board** (échiquier) qui propose les méthodes suivantes :

- *putQueen(self, square)* qui place une dame sur la case square, avec square une chaîne de caractères de la forme 'a1'.
- *hasQueen(self, square)* qui retourne True si la case square est occupée par une reine, False sinon.
- *removeQueen(self, square)* qui retire la dame de la case square.

Listing 2 – code d'utilisation de la classe Board

```
board = Board() # l'\'echiquier est vide
board.hasQueen('a1') # False
board.putQueen('a1')
board.hasQueen('a1') # True
board.removeQueen('a1')
board.hasQueen('a1') # False
```

Listing 3 – code de manipulation des chaînes de caractères

```
b8 = 'b8'
row = int(b8[1]) # 8
column = int(ord(b8[0]) - 96) # 2
a1 = chr(97) + str(1) # 'a1'
```

Question 2. Programmer la classe Board et faire en sorte que le code du listing 2 puisse être exécuté. Le code du listing 3 vous aide à effectuer les manipulations de chaînes de caractères.

Question 3. Programmer la fonction *queen_legal_moves(square)* qui retourne l'ensemble des cases sur lesquelles une reine peut bouger si elle est placée sur la case **square** et en considérant un échiquier vide. La fonction doit retourner toutes les cases de sa ligne, de sa colonne, ainsi que les deux diagonales qui passent par sa case. Enfin, la case d'origine ne doit pas appartenir à l'ensemble retourné.

Question 4. Modifier la fonction *check_solution8queens(sol)* pour qu'elle prenne en entrée un échiquier (un objet instance de la classe Board). La fonction devient alors : *check_solution8queens(board)*. Vous vous appuierez sur la fonction *queen_legal_moves(square)* pour savoir si l'échiquier passé contient une solution correcte au problème des huit dames.

Tournez la page S.V.P.

La dame et les autres pièces

On considère la classe **Queen** qui représente une dame. Dans un premier temps cette classe ne contient qu'une seule méthode : *legal_ moves(self, square)*. Cette méthode retourne la liste des cases sur lesquelles une dame peut se déplacer si elle est placée sur la case **square** et si on considère un échiquier vide. Le code de cette méthode est donc le même que la fonction *queen_ legal_ moves(square)*.

Question 5. La conception de la classe **Queen** fait le choix que la position des pièces est connue par classe **Board** (et non par **Queen**). Discuter de l'intérêt de ce choix de conception (avantages et inconvénients).

Déplacement des pièces

Pour rendre le jeu plus intéressant, on va ajouter d'autres pièces : le roi, la tour et le fou. Quelques explications sur le déplacement de ces pièces aux échecs sont données ci-dessous. Notez que des explications sur les autres pièces du jeu d'échecs (le cavalier ou le pion), ne sont pas données car nous ne les considérerons pas dans la suite du sujet. Nous ne traiterons également pas des règles comme le pat ou la prise en passant.

Question 6. Programmer en Python les classes **Queen** (Dame), **King** (Roi), **Rook** (Tour) et **Bishop** (Fou). On ne considère pas le cavalier et le pion. On se contentera d'implémenter les mouvements possibles des pièces, en s'appuyant sur les explications données préalablement. Ces classes proposent la même méthode *legal_ moves(self, square)* qui donne la liste des cases sur lesquelles une pièce peut bouger à partir d'une case de départ et en considérant un échiquier vide. Il est attendu à ce que vous utilisiez l'héritage pour définir ces classes et faire en sorte qu'il y ait peu de redondance de code.

Question 7. Modifier la classe **Board** et remplacez les méthodes *putQueen*, *hasQueen* et *removeQueen* par des méthodes *put*, *has* et *remove* pour toutes les pièces considérées (Queen,

> Le fou, la tour et la dame sont des pièces à longue portée (ou pièces de lignes) : elles peuvent se déplacer le long de lignes. Les fous se déplacent toujours sur les cases d'une même couleur, en diagonale. La tour est capable de se mouvoir en ligne droite, verticalement, horizontalement, sur un nombre quelconque de cases inoccupées (voir figure 3).
> La dame est capable de se mouvoir en ligne droite, verticalement, horizontalement, et diagonalement, sur un nombre quelconque de cases inoccupées (voir figure 3). Le roi se déplace d'une seule case à la fois dans toutes les directions. Les pièces (excepté le pion) capturent une pièce adverse qui se trouve sur leur trajectoire, sans pouvoir aller au-delà. La pièce qui capture prend la place de la pièce capturée, cette dernière étant définitivement retirée de l'échiquier.

FIGURE 2 – Explications sur les déplacements des pièces

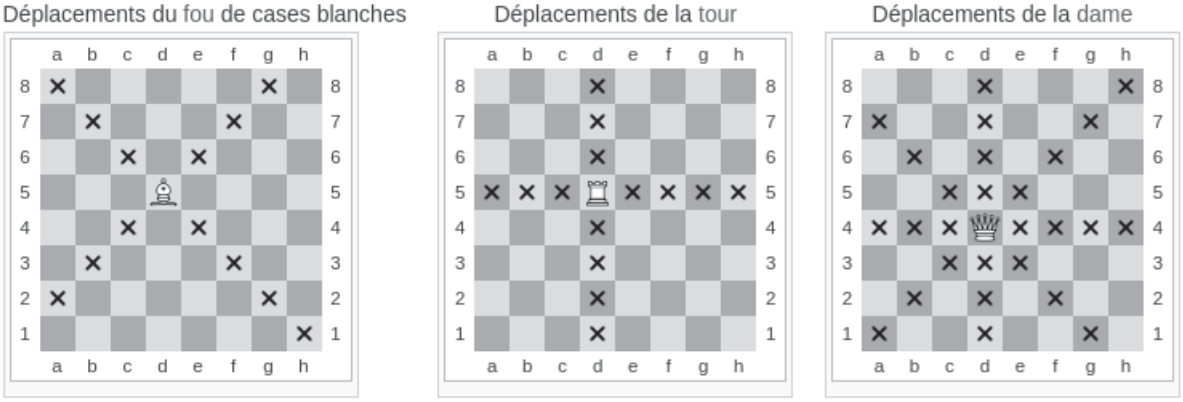

FIGURE 3 – Déplacements Fou, Tour et Dame

King, Rook et Bishop). Vous ajouterez également la méthode *get(self,square)* qui retourne la pièce positionnée à la case square, ou None si aucune pièce n'y est positionnée. Il est attendu que votre code soit le plus générique possible et qu'il y ait le moins de redondance et le plus de réutilisation. Charge à vous de proposer une conception Orientée Objet adaptée et de bien définir les arguments de ces méthodes. Une fois votre classe programmée, le code du listing 4 devra pouvoir être exécuté.

Listing 4 – code d'utilisation de la classe Board

```
board1 = Board()
board1.put('a1', Queen())
board1.put('b3', Rook())
board1.has('a2') # False
queen = board1.get('a1') # Queen
```

Les problèmes similaires au problèmes des huit dames

On considère la classe **Problem8Queens** qui représente le problème des 8 dames. Cette classe a les méthode suivantes :

- *check_solution8queens(self, board)* qui permet de vérifier que la solution board est valide.
- *propose_solution(self, board)* qui permet à un joueur de proposer une solution au problème. Cette méthode fait appel à la méthode *check_solution8queens(self, board)* pour vérifier la solution proposée. De plus, elle mémorise dans un tableau toutes les solutions proposées par les joueurs, que celles-ci soient valides ou non.
- *statistiques(self)* qui écrit un rapport sur la sortie standard. Ce rapport précise combien de solutions ont été proposées et combien sont valides.

Question 8. Programmer la classe **Problem8Queens**. Vous prendrez soin d'exploiter les méthodes de la classe **Board** et les méthodes des classes représentant les pièces. Une fois votre classe programmée, le code du listing 5 devra pouvoir être exécuté.

Listing 5 – code d'utilisation de la classe Problem8Queens

```
board1 = Board()
board1.put('a1', Queen())
board1.put('b3', Queen())
problem = Problem8Queens()
problem.propose_solution(board1)
problem.statistiques() # affiche sur la sortie standard "1 solution, 0 valide"
```

On souhaite maintenant proposer différents types de problèmes : le problème des huit dames, le problème des huit tours, le problème des 4 fous et des 4 tours, etc.

La classe **ProblemFactory** propose une méthode *create_problem* permettant de créer des problèmes de différents types (8 dames, 8 fous, etc.). A noter qu'il est également possible créer des problèmes avec une combinaison de différents types de pièces (exemple : 4 fous et 4 tours).

Question 9. Décrire la classe **ProblemFactory** (sans la programmer en Python) en prenant soin de bien définir les paramètres de la méthode *create_problem* permettant de créer des problèmes de différents types. Vous expliquerez comment cette méthode permet de créer a minima le problème des 8 dames, le problème des 8 tours et le problème des 4 fous et des 4 tours. Vous décrirez également le lien avec la classe **Problem8Queens** et les modifications qu'il faut y apporter. Enfin vous décrirez les qualités de votre conception en termes de réutilisation de code.

Résolution de problème

On considère qu'un algorithme résout un problème du type problème des 8 dames s'il retourne toutes les solutions à ce problème. De manière générique, on appelle *solve* les fonctions qui implémentent un tel algorithme.

Question 10. Programmer une fonction Python qui : (1) génère une instance de la classe **Board** contenant 8 dames placées aléatoirement, (2) vérifie que cette instance est une solution au problème des 8 dames en appelant la méthode *check_solution8queens(self,board)* d'un objet instance de la classe **Problem8Queens**, et (3) s'arrête lorsqu'une solution valide a été trouvée en imprimant sur la sortie écran le résultat trouvé. Vous discuterez de la qualité de la solution consistant à exploiter cette fonction pour proposer une fonction *solve_random* qui résout le problème des 8 dames.

Question 11. Programmer une fonction Python *solve_walker* qui parcoure l'espace des **board** contenant 8 dames et qui s'arrête dès qu'ont été trouvées x **board** solutions au problème des 8 dames ($0 < x \leq 92$) en imprimant sur la sortie écran chaque solution trouvée. Sachant qu'il existe 92 solutions au problème des 8 dames, on peut dire que cette fonction propose un algorithme qui résout le problème des 8 dames. Est-ce que la connaissance du nombre de solutions (92) est nécessaire ?

Question 12. Programmer une fonction *solve_matrix* qui résout le problème des 8 dames en exploitant les matrices de *permutation*. En effet, comme les dames attaquent le long des lignes, des colonnes et des diagonales, cela équivaut à une matrice de permutation d'ordre 8 dans laquelle la somme de chaque diagonale est au plus égale à 1. On rappelle qu'une matrice de *permutation*

est une matrice carrée qui vérifie les propriétés suivantes : les coefficients sont 0 ou 1 ; il y a un et un seul 1 par ligne ; il y a un et un seul 1 par colonne.

Question 13. À ce stade, nous avons au moins trois implémentations candidates pour calculer l'ensemble des solutions du problème des huit dames : *solve_walker*, *solve_random*, et *solve_matrix*. Ces implémentations ont des qualités différentes mais en toute rigueur, l'ensemble des solutions retourné par les trois implémentations devrait donner le même résultat. Montrer en Python comment cette propriété peut être automatiquement vérifiée avec un programme.

Restructuration et difficulté des problèmes

On souhaite que n'importe quel développeur puisse proposer sa fonction *solve*. L'idée étant de pouvoir disposer de plusieurs fonction *solve* proposées par des développeurs différents.

Question 14. Décrire la restructuration de votre application permettant à un développeur d'intégrer sa fonction *solve* dans votre conception. Vous devez prendre soin de bien décrire toutes les modifications à effectuer sur le code des classes existantes. Vous expliquerez aussi ce que doit faire le développeur pour intégrer sa fonction. Enfin, vous montrerez comment choisir une fonction *solve* qui a été proposée, et comment l'exécuter pour trouver toutes les solutions à un problème.

Question 15. On souhaite générer des problèmes d'échecs difficiles. La notion de difficulté est définie ainsi : un problème d'échecs est difficile si (1) les trois implémentations précédentes en charge de trouver toutes les solutions mettent plus de 10 minutes ; et si (2) le nombre de solutions est inférieur à 10. Le problème des huit dames, par exemple, n'est pas difficile (car le nombre de solutions, 92, est supérieur à 10). Proposer un algorithme pour générer et trouver de tels problèmes difficiles. Expliquer comment cet algorithme peut s'intégrer dans la conception des questions précédentes (classe **ProblemFactory**, fonction *solve*, etc.)

Pour mesurer le temps d'exécution d'un programme en Python, vous vous appuierez sur le module *time*. Le listing 6 illustre le fonctionnement de *time*.

Listing 6 – code d'utilisation du module time

```
1 import time
2 start = time.time() # avant l'appel d'un programme
3 # execution du programme
4 # ...
5 end = time.time() # fin du programme
6 duration = end - start # la duree de l'execution en seconde (de type float)
```

Partie d'échecs

On souhaite maintenant étendre notre conception et implémentation précédente pour permettre de jouer une vraie partie d'échecs avec notre application.

Question 16. Modifier votre code pour prendre en compte la couleur des pièces. Vous ne recopierez pas tout votre code et vous décrirez uniquement le code modifié.

Question 17. Dans la classe **Board**, programmer la méthode *move(self, from, to)* qui déplace la pièce de la case **from** vers la case **to**. Cette méthode vérifiera que le déplacement est légal (via la méthode *legal_ move* des pièces) et qu'il n'y a pas de pièces entre la case **from** et la case **to** (pour rappel on ne considère pas le cavalier et le pion). La méthode vérifiera aussi que la case d'arrivée ne doit pas contenir une pièce de la même couleur que la pièce déplacée. Enfin, il ne devra pas être possible de déplacer deux fois de suite des pièces de la même couleur. Vous veillerez à ce que le code de la méthode *move* réutilise un maximum de code existant de la classe **Board** et des classes des pièces.

Nous considérons maintenant la classe **Game** qui représente une partie d'échecs. A l'initialisation, une partie contient un Board avec les pièces en position initiale. Il sera alors possible de jouer le premier coup des blancs. Puis, alternativement, il sera possible de jouer les coups des noirs et des blancs jusqu'au mat.

Question 18. Proposer une conception générale de la classe **Game** permettant aux joueurs de déplacer les pièces jusqu'au mat. Certaines particularités du jeu d'échecs (prise en passant, pat, fin de partie, etc.) non décrites dans l'extrait (cf figure 2, page 4) ne doivent pas être considérées ici. On ne détectera donc pas le mat. Donner les signatures des méthodes que vous jugez nécessaires (pas le code) pour que les joueurs puissent jouer une partie. Préciser par du texte les propriétés de cette classe (sans donner le code de la méthode *__init__*).

Question 19. Illustrer les 4 premiers coups échangés entre deux joueurs (2 coups chacun) à l'aide d'un diagramme décrivant la séquence des échanges de messages entre les objets de l'application. L'objectif est d'illustrer graphiquement combien d'objets participent à ces échanges, comment ces objets sont créés et reliés entre eux, et dans quel ordre se font les échanges. Votre diagramme devra alors faire apparaître graphiquement : tous les objets impliqués, tous les messages échangés, et l'ordre de ces messages

Le roque

On veut maintenant supporter et implémenter le *roque*. Des explications, issues et adaptées de Wikipedia, sont données dans l'encadré ci-dessous.

Question 20. Implémenter une fonction en Python qui implémente le mouvement du roque en vérifiant préalablement si le roque est possible. On fera l'hypothèse que le joueur indiquera la case de départ du roi et la case d'arrivée (le déplacement de la tour étant déduit). Expliquer comment vous avez organisé votre code notamment pour vérifier la possibilité du roque.

Le roque est un déplacement spécial du roi et d'une des tours au jeu d'échecs. Le roque permet, en un seul coup, de mettre le roi à l'abri tout en centralisant une tour, ce qui permet par la même occasion de mobiliser rapidement cette dernière. Il s'agit du seul coup légal permettant de déplacer deux pièces, sans respecter le déplacement classique du roi et de la tour de surcroît. On distingue le petit roque et le grand roque. Dans le cas du petit roque, le roi blanc se déplace de deux cases en direction de la tour, et la tour se place immédiatement à gauche du roi. La figure 5 donne une illustration.
Le grand roque s'effectue de la même manière que le petit roque : le roi avance de deux cases en direction de la tour (en a1 ou en a8), et cette dernière se place de l'autre côté du roi. La figure 6 illustre le principe.
Une particularité importante du roque est que plusieurs conditions doivent être respectées :

- toutes les cases qui séparent le roi de la tour doivent être vides. Par conséquent, il n'est pas permis de prendre une pièce adverse en roquant ;
- ni le roi, ni la tour ne doivent avoir quitté leur position initiale. Par conséquent, le roi et la tour doivent être dans la première rangée du joueur ;
- chaque camp ne peut roquer qu'une seule fois par partie ; en revanche, le roi peut déjà avoir subi des échecs, s'il s'est soustrait à ceux-ci sans se déplacer.
- aucune des cases (de départ, de passage ou d'arrivée) par lesquelles transite le roi lors du roque ne doit être contrôlée par une pièce adverse. Par conséquent, le roque n'est pas possible si le roi est en échec.

FIGURE 4 – Explications du roque

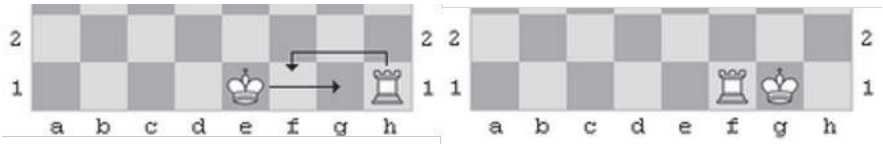

FIGURE 5 – Petit roque (avec les pièces blanches)

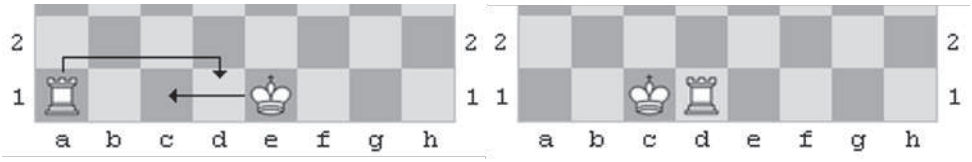

FIGURE 6 – Grand roque (avec les pièces blanches)

Tournez la page S.V.P.

Problème des mats en N coups

L'application **ChessMate** ne porte pas bien son nom puisque, en l'état, les problèmes d'échecs et mat ne sont pas supportés. On se propose d'y remédier.

La classe **MatInNMoves** représente un problème de type mat en N coups. Un tel problème est défini par une position initiale des pièces sur l'échiquier, une couleur (blanc ou noir) pour laquelle le problème est posé, et un nombre de coups maximum pour que cette couleur mette l'autre couleur en position de mat (c'est-à-dire dans l'impossibilité d'empêcher la prise de son roi).

Question 21. Proposer les méthodes de cette classe (pas le code) permettant à un utilisateur d'essayer de résoudre un tel problème. A noter que dans le cadre spécifique des mats en N coups, c'est la même entité qui manipule à la fois le joueur avec les pièces blanches et avec les pièces noires. Vous ferez en sorte que la classe **MatInMoves** réutilise un maximum la classe **Game**, et vous expliquerez la façon dont la vérification du problème se fait.

Réflexion sur la conception

Question 22. Il existe différentes façons de représenter l'échiquier dans la classe **Board**. Cela peut se faire notamment à l'aide d'un tableau comme c'est le cas dans le code donné du listing 1. Proposer une autre façon de représenter l'échiquier et discuter de ses avantages et de ses inconvénients (en espace et en temps).

Question 23. On souhaite généraliser le problème des huit dames (et uniquement ce problème) et faire en sorte que l'échiquier soit de $N \times N$ cases (N étant un nombre positif donné lors de la création du Board). Discuter de l'impact de cette généralisation sur la conception réalisée jusqu'à présent. Vous préciserez quels sont les impacts sur les classes et sur leurs méthodes.

Partie II. Web

On souhaite réaliser une interface graphique HTML, CSS et JavaScript permettant de jouer à des problèmes échiquéens.

L'application Web est composée de deux pages. La première page permet au joueur de sélectionner le problème. L'interface voulue est présentée par la figure 7.

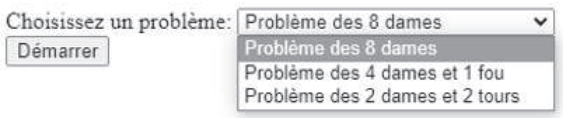

FIGURE 7 – Première page de l'application Web

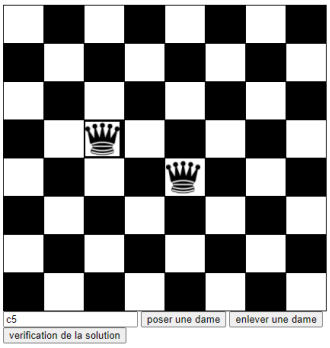

FIGURE 8 – Deuxième page de l'application Web pour le problème des 8 dames

La deuxième page permet de jouer (résoudre le problème proposé). La figure 8 présente l'écran pour tenter de résoudre le problème des 8 dames. On y voit l'échiquier avec ses 64 cases. Une première rangée de boutons permet de saisir les coordonnées d'une case (sur l'exemple : 'c5' et 'e4'), de déposer ou d'enlever une dame. Enfin, un dernier bouton permet de demander la vérification de la solution qui sera réalisée au niveau du serveur web. Le code HTML de la deuxième page web vous est partiellement donné pour le problème des 8 dames (voir listing 7).

Listing 7 – page HTML du problème des 8 dames

```
<html>
<head>
    <title>Chess Board</title>
    <style>
        table {
            border-collapse: collapse;
            border: 1px solid black;
        }
        td {
            width: 50px;
            height: 50px;
            border: 1px solid black;
        }
        img {
            width: 45px;
            height: 45px;
        }
        .black {
            background-color: black;
        }
        .white {
            background-color: white;
        }
    </style>
</head>
<body>
    <table id="board">
    </table>
    <div>
        <input type="text" id="square" value="e4" />
        <input type="button" id="putQueen" value="poser une dame" />
        <input type="button" id="removeQueen" value="enlever une dame" />
    </div>
    <div>
        <input type="button" id="solve" value="verification de la solution" />
    </div>
</body>
</html>
```


Tournez la page S.V.P.

Question 24. Programmer la première page. Celle-ci contient un formulaire permettant au joueur de choisir le type de problème qu'il veut résoudre. Ce formulaire soumet une requête HTTP GET vers "/choose_problem". Il contient un seul champ nommé "problem". Vous proposerez une programmation en HTML pur de l'ensemble de cette page (i.e., sans JavaScript ou CSS).

Question 25. Dans la deuxième page, programmer un script JavaScript permettant d'afficher un échiquier à l'ouverture de la page. L'échiquier est un tableau composé de 8 lignes (balises TR) et de 64 cellules en tout (balises TD). Vous ferez en sorte que chaque cellule utilise le style proposé pour la couleur des cases. De plus, chaque cellule doit avoir un identifiant unique. Votre code sera inséré dans la page HTML de la deuxième page web à l'aide de la balise proposée dans le listing 8. Les premières lignes du code JavaScript sont déjà écrites, il vous reste à écrire le code pour afficher l'échiquier.

Listing 8 – script pour afficher l'échiquier

```
<script>
    let board = document.getElementById("board");
    for (let i = 8; i > 0; i--) {
        //TODO
    }
</script>
```

Question 26. Dans la deuxième page, programmer un script JavaScript permettant de déposer une dame sur la case choisie. Il faudra alors ajouter une image (balise IMG) à la cellule choisie avec l'attribut src de la balise IMG étant égal à "queen.jpg". La première ligne du code JavaScript est déjà écrite (voir listing 9), il vous reste à compléter ce code.

Listing 9 – script pour déposer une dame

```
<script>
    let putQueenButton = document.getElementById("putQueen");
    //TODO
</script>
```

Question 27. Programmer un script JavaScript permettant de retirer une dame de la case choisie. Vous ferez en sorte de minimiser la redondance avec le code de la question précédente. La première ligne du code JavaScript est déjà écrite dans le listing 10, il vous reste à compléter ce code.

Listing 10 – script pour supprimer une dame

```
<script>
    let removeQueenButton = document.getElementById("removeQueen");
    //TODO
</script>
```

La vérification d'une solution s'effectue en envoyant une requête HTTP vers un serveur web (note : vous n'avez pas à proposer d'implémentation du serveur web). La requête de vérification est une requête GET vers **/solve8queens** et qui a un paramètre dont le nom est **solution** et dont la valeur est une chaîne de caractères décrivant les positions des dames en séparant chaque position par le caractère '-'. Par exemple, si l'adresse du serveur est http ://www.serveur-chess.fr, envoyer la requête GET http ://www.serveur-chess.fr/solve8queens ?solution=e1-b2-c3-a4-h5-d7-f8-h6 permet de demander la vérification de la solution : ('e1', 'b2', 'c3', 'a4', 'h5', 'd7', 'f8', 'h6').

Si la réponse à la requête est positive (code de retour : 200), le script doit ouvrir une alerte précisant que la solution est valide. Sinon, une alerte précisera que la solution n'est pas valide. On rappelle que la fonction JavaScript "alert(message)" permet de lancer une alerte à l'utilisateur.

Question 28. Programmer ce script JavaScript permettant de soumettre une solution au problème des 8 dames. La première ligne du code JavaScript est déjà écrite dans le listing 11, il vous reste à compléter ce code.

Listing 11 – script pour envoyer une solution

```
<script>
    let solveButton = document.getElementById("solve");
    //TODO
</script>
```

Question 29. Discuter des avantages et inconvénients de réaliser certaines validations de la solution proposée avant d'envoyer une demande de validation au serveur (on considère que le calcul de la vérification est trop long pour se faire intégralement sur le navigateur).

Partie III. Base de données

Pour améliorer l'application **ChessMate** il a été convenu de mettre à disposition une base de données qui contient différents problèmes d'échecs.

Chaque problème a un titre et une description. De plus, certains problèmes sont accompagnés d'une solution déjà trouvée et établie que les utilisateurs pourront consulter. D'autres problèmes, ouverts (‘open‘ en anglais), n'ont pas de solution associée. Enfin, pour certains problèmes, un programme de vérification (‘checking_procedure‘ en anglais) est éventuellement associé et permet de vérifier si la solution proposée au problème est correcte.

Pour commencer la construction de cette base de données, il a été convenu d'utiliser le format **YAML**. qui est un format de représentation de données clé/valeur et arborescent. Un exemple de données en cours de construction est donné ci-dessous :

Listing 12 – exemple de la base

```
chessmate:
    - description: "Le but du probl\‘eme des huit dames est de placer..."
      version: "1.0.0"
      title: "8 queens problem"
      type: "all_solutions"
      contact:
        name: "Elisabeth Harmon"
        email: "XXX@agreg-info.org"
      problem: "QQQQQQQQ"
      checking_procedure: "https://chessmate.org/8queens"
      solutions: [[8, 5, 20, 25, 42, 59, 54, 39], [0, 14, 20, 31, 33, 43, 53, 58]]


    - description: "Votre objectif est de placer 4 dames et 1 fou..."
      title: "4 Queens + Bishop problem"
      type: "one_solution"
      open: true
      contact:
        name: "Judit Polkarov"
```


Tournez la page S.V.P.

```
20          email: "YYY@agreg-info.org"
21        problem: "QQQQB"
22        checking_procedure: "https://chessmate.org/4QueensBishop"
23
24      - description: "Votre objectif est de placer 2 dames et 2 tours sur un plateau 6↘
            par 6..."
25        version: "0.1"
26        title: "2 Queens + 2 Rooks problem"
27        type: "one_solution"
28        open: true
29        contact:
30          name: "Judit Polkarov"
31          email: "YYY@agreg-info.org"
32        problem: "QQRR"
33        chessboard: 36
34        checking_procedure: "https://chessmate.org/2QueensRooks36"
35
36      - description: "Mate in 2"
37        version: "0.1"
38        title: "Mate in 2"
39        type: "check_mate"
40        moves: 3
41        turn: "white"
42        contact:
43          name: "Garry Agreg"
44          email: "XXYY@agreg-info.org"
45        problem: "r1b2r1k/ppp3p1/2n2q1N/7Q/2BPp3/2P5/P4PPP/R1B3K1 w - - 1 16"
46        solutions: "Nf7+ Kg8 Qh8#"
```

L'exemple YAML fourni spécifie 4 problèmes. Les trois premiers problèmes sont des "puzzles", avec respectivement le problème des huit dames, le problème des 4 dames et du fou, et le problème des 2 dames et des deux tours sur un échiquier 6*6.

Le quatrième est un problème d'échecs et mat : il faut trouver une série de 3 coups ('moves' en anglais) en commençant par un coup des pièces blanches (cf turn et white).

La description des puzzles ou les problèmes de mat partagent un certain nombre de caractéristiques, mais exhibent également des informations spécifiques. On peut catégoriser ainsi les informations caractérisant les problèmes :

- *ProblemDescription* : description du problème, version du problème, titre du problème, type de problème (échecs et mat, puzzle), si le problème est ouvert, etc. Un problème a nécessairement une description qui est une chaîne de caractères, possiblement une version qui est une chaîne de caractères, un titre via title. La notion de turn n'a aucun sens pour les problèmes de type puzzle et donc n'est pas spécifiée pour les 3 problèmes de l'exemple. Quand chessboard n'est pas spécifié, on considère par défaut que l'échiquier est 8*8 et contient 64 cases. Pour les "puzzles", on distingue deux types de problèmes : le cas où il faut trouver une seule solution (one_solution), et le cas où il faut trouver toutes les solutions (all_solutions). Toujours pour les puzzles, un problème est considéré comme ouvert (open) si une ou des solutions ne sont pas connues et donc renseignées. Pour les "échecs et mat", la valeur de type est nécessairement égale à check_mate ;
- *Author* : des informations sur l'auteur du problème avec son nom (name) et son email (email) ;

- *Solution* : une ou des solutions si elles existent, spécifié dans ‘solutions‘ avec une chaîne de caractères et une syntaxe bien particulière selon que ce soit un "puzzle" ou un "échecs et mat".

On souhaite maintenant remplacer l'utilisation de **YAML** par un système de gestion de bases de données (SGBD) relationnel.

Autrement dit, nous souhaitons migrer le document YAML dans un SGBD relationnel.

Question 30. Proposer un modèle relationnel avec au moins trois tables : "ProblemDescription", "Author", "Solution". Pour chaque table, donner les noms de colonne ainsi que leurs types.

En plus de la description des tables, vous préciserez vos choix en répondant aux quatre questions ci-dessous :

Question 31. Comment avez-vous représenté l'attribut "type" ?

Question 32. Comment avez-vous représenté l'optionalité de la ‘version‘ d'un problème ?

Question 33. Comment avez-vous organisé les informations spécifiques à un "échecs et mat" et à un "puzzle" ?

Question 34. Dans ce schéma relationnel, quelles sont les clés primaires de chaque table ? Qu'existe-t-il comme clés étrangères ?

Pour illustrer le fonctionnement de votre base de données :

Question 35. On veut ajouter le problème échec et mat "Mate in 2" de l'exemple YAML ci-dessus. Donner le(s) ordre(s) d'ajout nécessaire(s) dans les tables.

En vous appuyant sur vos tables, écrire en SQL :

Question 36. une requête retournant tous les problèmes "ouverts" (correspondant à `open` dans le YAML).

Question 37. une requête retournant des problèmes, de type puzzle, avec uniquement 8 * 8 cases.

Question 38. une requête retournant le nombre de problèmes qui ont des solutions ou qui sont des problèmes de type "échecs et mat".

On souhaite identifier tous les problèmes de type "puzzle" qui mettent en jeu au moins un fou (‘Bishop‘ en anglais).

Question 39. Est-ce possible de le faire uniquement avec une requête SQL ? Justifier.

Pour y parvenir on se propose de le faire en deux temps :

- via une requête SQL qui retourne tous les problèmes de type "puzzle" (excluant de fait les échecs et mat) et en particulier l'information problem qui décrit le problème avec une syntaxe spécifique. Par exemple, pour le problème numéro 2 de l'exemple YAML, problem : "QQQQB" spécifie qu'il y a 4 dames à positionner (Q pour Queen) et 1 fou à positionner (B pour Bishop) ;
- via un programme Python qui analysera le contenu de 'problem' pour ne garder que les problèmes avec au moins un fou.

Question 40. En vous appuyant sur l'API Python sqlite3 dont le principe de fonctionnement est décrit ci-dessous, programmer une telle solution en Python. Deux modifications de Listing 13 sont à réaliser : (1) adapter la ligne 4 pour écrire la requête SQL ; (2) adapter la ligne 6, possiblement en écrivant plusieurs lignes de code avant ou après, pour analyser le contenu et ne garder que les problèmes qui utilisent au moins un fou.

Les lignes 1, 2 et 3 permettent de se connecter à la base de donnée et n'ont pas d'intérêt majeur. La ligne 4 permet d'itérer sur chaque ligne (row) de la requête. La ligne 5 permet d'obtenir un dictionnaire avec clé/valeur de la ligne, la clé étant le nom de la colonne (dans cet exemple fictif : date, type, price, city). Ainsi, ligne 6, il est possible de traiter ce dictionnaire et d'analyser le contenu de la ligne (dans cet exemple fictif, c'est une simple impression écran avec print).

Listing 13 – API Python sqlite3

```
1 import sqlite3
2 con = sqlite3.connect('example.db') # nom de la base
3 cur = con.cursor()
4 for row in cur.execute('SELECT * FROM stocks ORDER BY price'): # requete SQL
5     r = dict(row) # dictionnaire cle/valeur
6     print(r)
```

Le résultat obtenu est :

Listing 14 – Résultat de la requête SQL

```
1 {'date': '2006-01-05', 'type': 'BUY', 'price': 35.14, 'city': 'Rennes'}
2 {'date': '2006-01-08', 'type': 'SELL', 'price': 5.42, 'city': 'Marseille'}
3 {'date': '2016-11-05', 'type': 'BUY', 'price': 18.94, 'city': 'Bordeaux'}
```

Question 41. Discuter des avantages et inconvénients de votre solution, basée sur une requête SQL puis un traitement programmatique, en termes de (1) maintenance et de (2) performance.

Partie IV. Architecture

Question 42. L'analyse des requêtes et de l'audience de **ChessMate** montre que le problème des 8 dames est très populaire et que de nombreux joueurs essayent de résoudre ce problème et proposent régulièrement des solutions. L'unique machine serveur qui vérifie ces solutions est

donc trop petite et ne supporte plus la charge. Proposer une infrastructure réseau permettant de déployer plusieurs serveurs. Cette infrastructure doit répartir les requêtes des joueurs (requêtes HTTP) sur les différents serveurs. Vous expliquerez de façon algorithmique et pratique comment ces requêtes sont réparties (comment orienter les requêtes vers les serveurs) et comment la charge est distribuée entre les serveurs (de manière équitable ? avec des effets de seuil ?).

Question 43. La solution proposée par la question précédente est intéressante mais elle cible essentiellement les pics de charge. En effet les serveurs déployés sont très souvent inexploités, la nuit par exemple lorsqu'aucun joueur ne propose de solutions. Proposer une infrastructure avec des serveurs virtuels. Vous expliquerez de façon algorithmique et pratique comment les serveurs virtuels se partagent la charge et comment ils sont démarrés et éteints. Vous préciserez combien de serveurs virtuels sont déployés (1) au déploiement de l'application et (2) lors d'un pic de charge.

Question 44. Dans l'application **ChessMate**, il est possible vérifier si une solution est bonne. Il est également possible de vérifier si *toutes* les solutions proposées sont bonnes. Plutôt que d'effectuer la vérification en séquentiel, sur une seule machine, solution après solution, on propose de distribuer la vérification des solutions sur plusieurs machines. Si une solution parmi toutes celles proposées n'est pas valide, alors la vérification s'arrête et retourne la solution erronée. Proposer une architecture permettant de distribuer une telle vérification de plusieurs solutions.

Question 45. L'approche distribuée de la question précédente peut potentiellement fournir des bénéfices en termes d'accélération du calcul et ainsi réaliser une vérification plus rapide. Cependant, cette approche a aussi des défauts car elle induit également des coûts computationnels et réseaux. Identifier tous les facteurs qui peuvent soit influer sur les bénéfices soit sur les coûts définis ci-dessus soit sur les deux à la fois. Discuter des compromis à trouver.