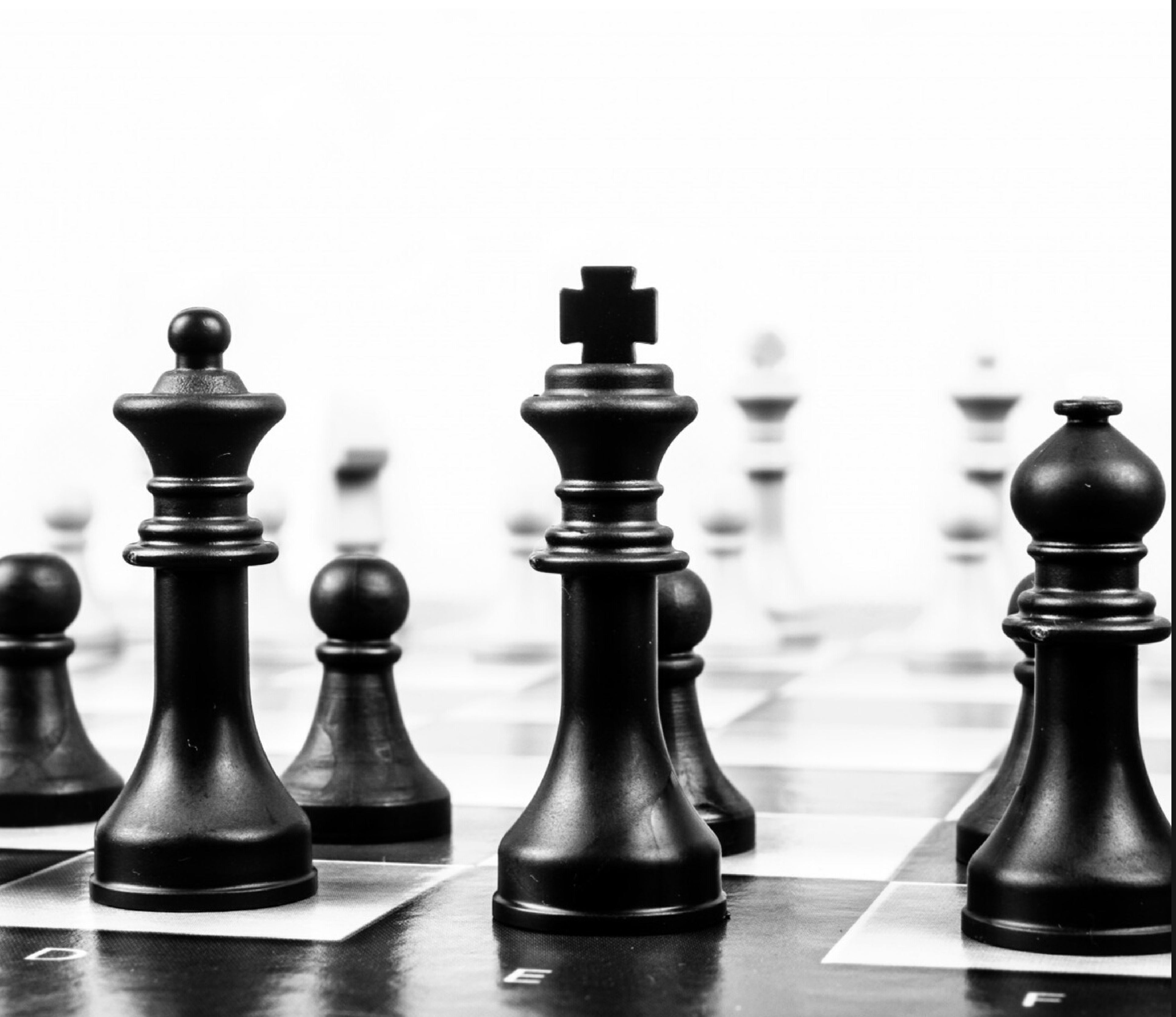
SAE : Exploration Algorithmique

par Levacher Ethan , Desperrois Lucas et Collin Ethan

Sommaire

Introduction

Nous sommes un groupe de trois étudiants qui nous sommes intéressées au problème des huit reines.

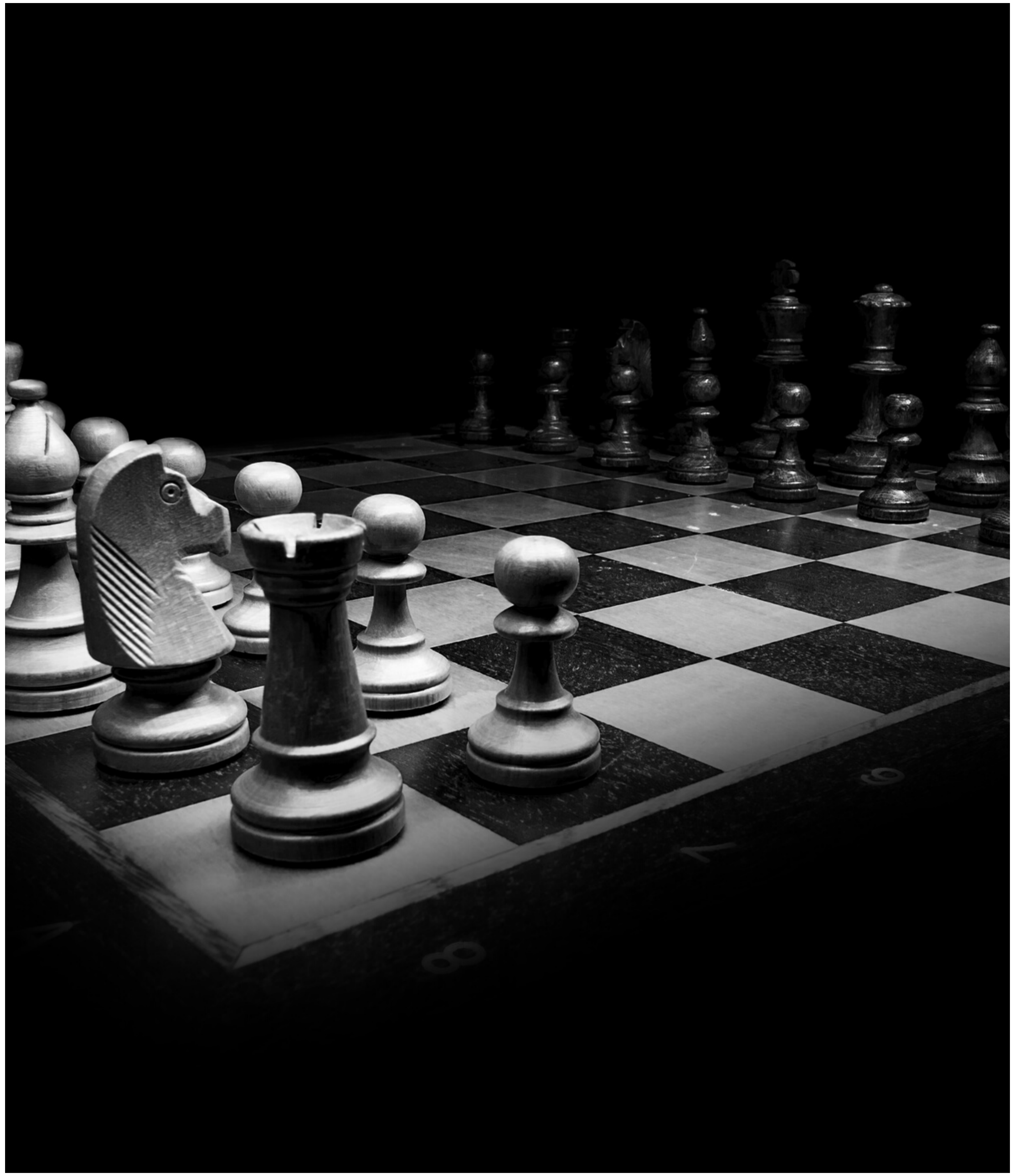


Problème

Le problème des huit reines est de placer huit reines dans un jeu d'échecs sur un échiquier de 8 × 8 cases sans que les reines puissent se menacer mutuellement, conformément aux règles du jeu d'échecs. Par conséquent, deux reines ne doivent jamais partager la même ligne, colonne, ou diagonale.

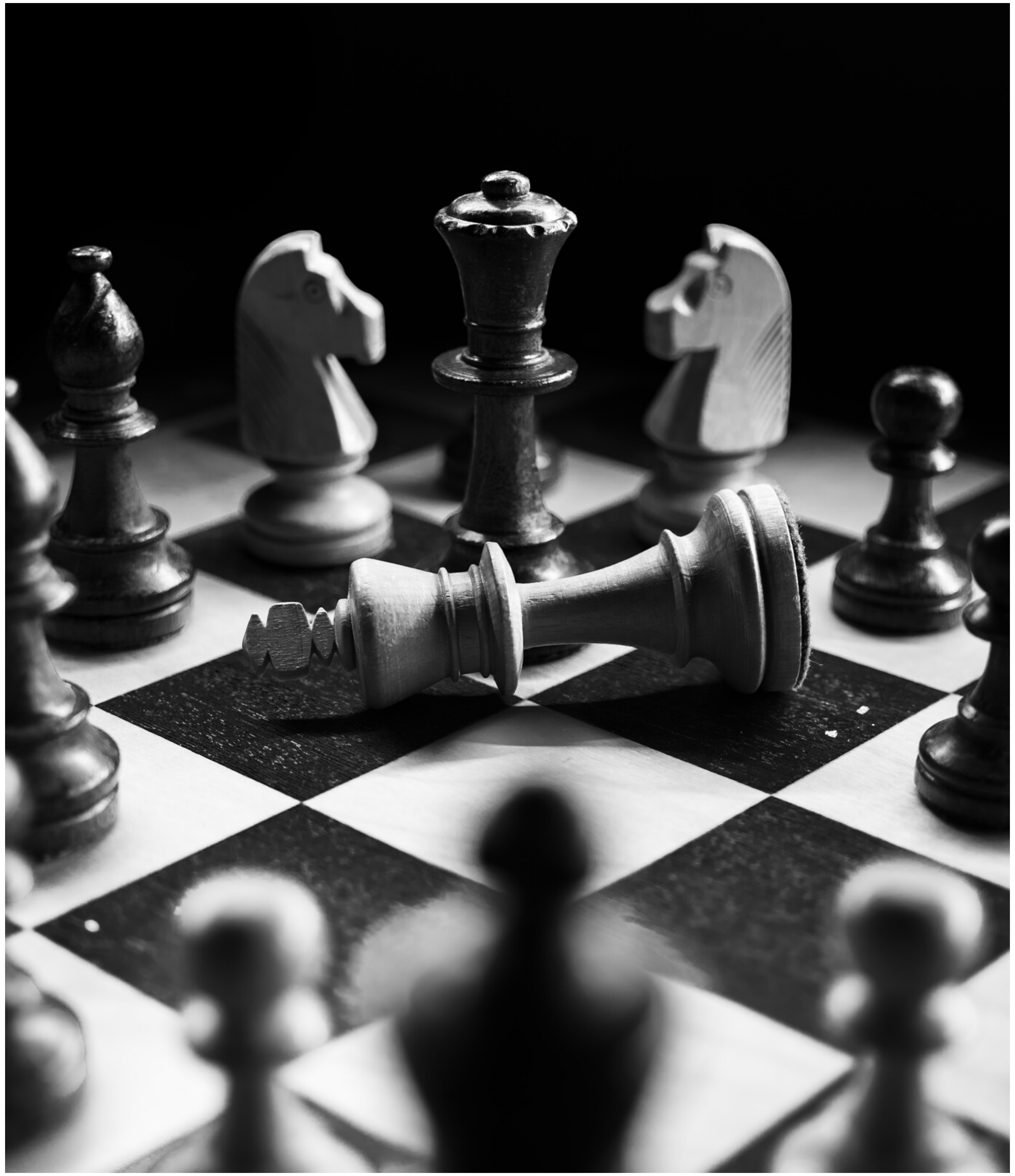
Particularités

Ce problème dispose de plusieurs particularités car on doit pouvoir résoudre le problème en considérant des échiquiers de tailles quelconque , ensuite de pouvoir les consulter.



Difficultés

Les difficultés rencontrées sont apparus lors de la création des algorithmes , plus particulièrement celui du BackTracking.
La difficultés était de montrer toutes les solutions possibles.



Solution

1ER ALGORITHME

Le 1er algorithme utilise
une approche de recherche
complète

2EME ALGORITHME

Le 2eme utilise le
BackTracking

3EME ALGORITHME

Le 3eme utilise le
BackTracking (d'une façon
différente)

Maquette et prototypes

Ces prototypes d'algorithmes permettent de répondre aux problèmes posées.

```
def createGrille(n):
    grille = [[0 for i in range(n)] for i in range(n)]
    return grille

def afficherGrille(s):
    for i in range(len(s)):
        print("|", end="")
        for j in range(len(s)):
            if (i == s[j]):
                print("1|", end="")
            else:
                print("0|", end="")
        print()

def possible(s, ligne, colonne):
    jouable = True
    l = 0
    while (l < ligne and jouable):
        c = s[l]
        jouable = (abs(ligne-l) != abs(colonne-c) and colonne != c)
        l = l + 1
    return jouable

def placerReine(ligne, n, sol):
    if ligne == n:
        liste_solutions.append(sol.copy())
```

```
        ok=False;
i=lig;
#verification  ouest
while(j-1>=0 and ok==True):
    j=j-1;
    if(grille[i][j]==1):
        ok=False;
j=col;
# vérification sud est
while(j+1<n and i+1<n and ok==True):
    i=i+1;
    j=j+1;
    if(grille[i][j]==1):
        ok=False;
i=lig;
j=col;
# vérification nord ouest
```

```
    for i in range(n):
        for j in range(n):
            # Ajouter toutes les positions possibles pour chaque reine
            placements[i].append((i, j))

    # Trouver toutes les combinaisons possibles de placements pour chaque reine
    all_combinations = cartesian_product(placements)

    # Trouver la première combinaison valide de placements de reines
    for combination in all_combinations:
        if is_valid(combination):
            liste_solutions.append(combination)
    print(liste_solutions)
    print(len(liste_solutions))
# Vérifier si une combinaison donnée de placements de reines est valide
def is_valid(combination):
    for i in range(len(combination)):
        for j in range(i+1, len(combination)):
            # Vérifier si deux reines ont une position en conflit
            if conflict(combination[i], combination[j]):
                return False
    return True

# Vérifier s'il y a un conflit entre deux reines placées sur les positions (i1, j
def conflict(pos1, pos2):
    i1, j1 = pos1
    i2, j2 = pos2
```

```python
### Author : COLLIN Ethan
### Algorithme d'exploration algorithmique pour résoudre
### Time : 31/03/2023

### Mon programme est fait en 2 dimension , en me servan


def solve_n_queens(n):
    #  Permet d'initialiser la liste des combinaisons po
    liste_solutions = []
    # Permet initialiser la liste de placements possible
    placements = [[] for i in range(n)]
    for i in range(n):
        for j in range(n):
            # Permet de ajouter toutes les positions pos
            placements[i].append((i, j))

    # Permet de Trouver toutes les combinaisons possible
    all_combinations = cartesian_product(placements)

    # Permet de Trouver la première combinaison valide d
    for combination in all_combinations:
        if is_valid(combination):
            liste_solutions.append(combination)
    print(liste_solutions)
    print(len(liste_solutions))
```

```python
# def is_valid permet de vérifier si u
def is_valid(combination):
    for i in range(len(combination)):
        for j in range(i+1, len(combi
            # Vérifie si 2 reines ont
            if conflict(combination[i
                return False
    return True

# def conflict Permet de vérifier s'il
def conflict(pos1, pos2):
    i1, j1 = pos1
    i2, j2 = pos2
    if i1 == i2 or j1 == j2 or abs(i1
        # vérifie si les positions son
        return True
    return False
```

```python
# def cartesian_product Permet de trouver l
def cartesian_product(lists):
    if len(lists) == 1:
        return [(item,) for item in lists[0
    else:
        result = []
        for item in lists[0]:
            # Grâce à la récursive cela per
            for rest in cartesian_product(l
                result.append((item,) + res
        return result

# Appele la fonction pour résoudre le probl
solve_n_queens(8)


### Cet algorithme utilise une approche de
### Cependant cet algorithme utilise beauco
### Cet algorithme ne permettra pas de réso
```

ÉTAPE 1

Définition d'une fonction générant toutes les combinaisons de dames possible sur une grille.

ÉTAPE 2

Définition d'une fonction permettant de verifier la validité d'une grille c'est à dire si les dames se mettent en danger ou non.

ÉTAPE 3

Définition d'une fonction permettant l'appelle des 2 fonctions précédentes afin de renvoyer uniquement les solutions valides.

1er Algorithme

2eme Algorithme

```
# - Type du résultat de la fonction : void
# - Résultat de la fonction : Affichage de l'échiquier pour 1 solution du prol

# - Param n : entier qui représente la taille de l'echiquier
# - Param grille : Liste de Liste qui représente l'échiquier

# - Fonctionnement : Utilisation d'une double boucle for pour afficher le tab

def affiche_plateau(n,grille):


    for i in range(n):                                  # double boucle for
        for j in range(n):
            if (grille[i][j]==1):
                print(" | dame" ,end='');
            else:
                print(" | ",grille[i][j]," ",end='');    #Affichage des barre
        print(" |",end='');


        print("\n");
    print("\n\n\n\n");

#---------Fonction valide_position(grille,col,lig)---------#
# - Type de résultat de la fonction : Booléen
# - Résultat de la fonction : retourne un booléen si la position de la n ème

# - Param grille : Liste de Liste qui représente l'échiquier
# - Param lig : entier qui représente la ligne où on test de poser la n-ème d
# - Param col : entier qui représente la colonne où on test de poser la n-ème

# - Fonctionnement_Fonction : on parcourt toute la grille pour tester si on p
# (lig,colonne) sans être menacées par les autres dames.


def valide_position(grille,lig,col):
    i=lig;
    j=col;
    ok=True;                                            # variable résultat;
    n=len(grille);
                                                        # le 1 sur la grille
                                                        #toutes les boucles w
                                                        #à la position(lig,co
```

```
    while(j-1>=0 and ok==True):
        j=j-1;
        if(grille[i][j]==1):
            ok=False;
    j=col;
    # vérification sud est
    while(j+1<n and i+1<n and ok==True):
        i=i+1;
        j=j+1;
        if(grille[i][j]==1):
            ok=False;
    i=lig;
    j=col;
    # vérification nord ouest
    while(j-1>=0 and i-1>=0 and ok==True):
        i=i-1;
        j=j-1;
        if(grille[i][j]==1):
            ok=False;
    i=lig;
    j=col;
    # verification nord
    while(i-1>=0 and ok==True):
        i=i-1;
        if(grille[i][j]==1):
            ok=False;
    i=lig;
    # verification nord est
```

```
def placer_reine(lig,echequier,n,l):
                                            #boucle qui parcourt toutes les colonnes d'une
    for col in range(n):

        echequier[lig][col]=1;
        if(valide_position(echequier,lig,col)):
                                            #si on a pas atteint la dernière ligne de l'éch
                                            #on appelle récursivement la fonction pour la l
            if(lig<n-1):

                placer_reine(lig+1,echequier,n,l);
            else:
                                            # lorsqu'on a fini de traité toutes les lignes
                                            # au compteur du nombre de solutions
                l[0]+=1;



        echequier[lig][col]=0;              #si la position tester n'est pas valide on met
    return l[0];                            # le compteur à l'indice 0 de la liste



#-------------Fonction main()---------------------------#
# - Type de resultat de la fonction : void
# - Resultat-de-la-fonction : affichage des nombres de solutions pour n dames mais aussi de
#
#
# - Fonctionnement-de-la-Fonction : elle propose de résoudre le probleme pur n dames et emp
# de résoudre le cas pour n<=3 car cela ne marche pas



def main():
    nb=int(input("donnez la taille de l'echiquier"));
    while(nb<=3):
        print("la taille de l'echiquier doit être supérieur à 3 sinon on ne peut pas placer
        nb=int(input("donnez la taille de l'échiquier"));
    print("le nombre de solutions pour le problème des dames avec un échiquier de",nb,"est
    rep=str(input("voulez-vous voir tous les échiquiers possibles pour le nombre de dames d
    if(rep=="1"):
        placer_reine_affichage(0,grille(nb),nb);
    else:
        print("aurevoir");



main();
```

ÉTAPE 1

Affichage de la grille avec une solution

ÉTAPE 2

Vérification de la position des dames avec un parcourt dans toutes les directions pour pouvoir valider l'affectation d'une dame à une position

ÉTAPE 3

Utilisation des deux fonction précédentes et mise en place du récursif pour le résoudre le problème des n-dames

```python
import time
"""
Fonction boolean renvoyant True si il est po
Paramètres:
    s: le tableau contenant les reines actue
    ligne: un entier qui correspond à la lig
    colonne: un entier qui correspond à la c
"""
def possible(s, ligne, colonne):
    jouable = True
    l = 0
    # On verifie pour chaque reine tant qu'i
    while (l < ligne and jouable):
        # on récupère la colonne de la reine
        c = s[l]
        # Verification le l est sur la même 
        # Et également si leurs colonne est 
        jouable = (abs(ligne-l) != abs(colon
        l = l + 1
    return jouable
```

```python
"""
Fonction récursive qui ajoute chaque combinaison d'éch
Paramètres:
    n: un entier correspondant à la taille de l'échequ
    solutions: une liste qui contiendra l'ensemble des
    sol: une liste représentant un échequier (utile po
    col: un entier correspondant la colonne dans laque
"""
def placerReine(n, solutions, sol=[-1 for i in range(8
    # Cas de base : Si nous avons n reines de placés :
    if col == n:
        solutions.append(sol.copy())
    else:
        ligne = 0
        # Boucle permettant de parcours chaque colonne
        while (ligne < n):
            # Verification si il est possible de jouer
            if possible(sol, col, ligne):
                # On place la reine
                sol[col] = ligne
                # On recommence l'étape avec la reine
                placerReine(n, solutions, sol, col+1)
                # On retourne en arrière en enelevant
                sol[col] = 0
            ligne = ligne + 1
```

```python
"""
Fonction qui permet de gérer l'appel
Utile pour calculer le temps d'execut
Paramètres:
    n: un entier correspondant à la t
"""
def huit_reines(n):
    liste_solutions = []
    placerReine(8, liste_solutions)
    return liste_solutions

# Calcul du temps d'execution :
t1 = time.time()
l = huit_reines(8)
t2 = time.time()-t1
print(f'Temps d\'execution de la fonc
```

ÉTAPE 1

Définition de la fonction permettant de verifier si il est possible de placer une nouvelle reines.

ÉTAPE 2

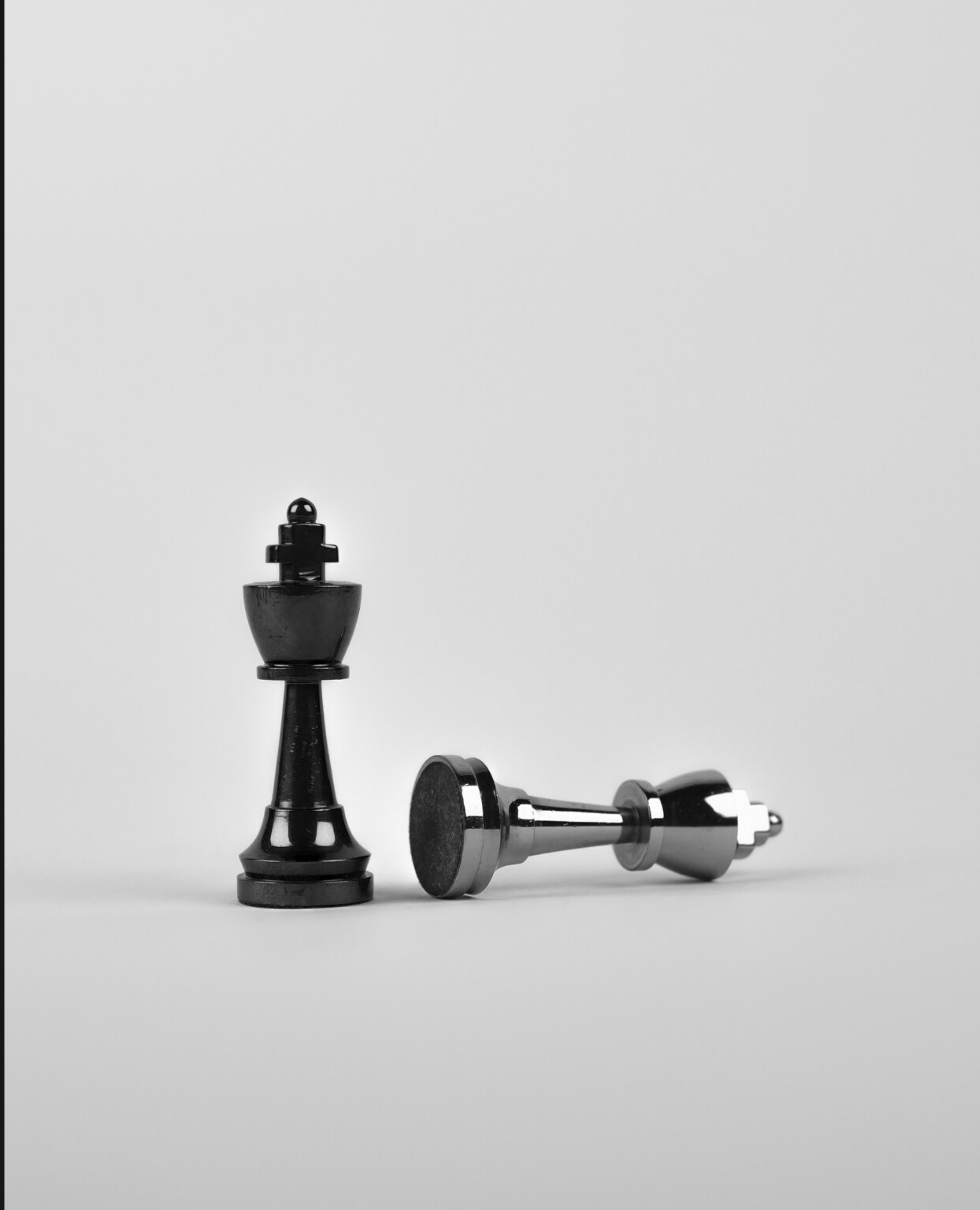
Création de la fonction récursif utilisant le backtracing afin de trouver l'entièreté des solutions aux problèmes.

ÉTAPE 3

Mis en place des fonctions précédentes dans une fonction principale et calcul du temps d'éxécutions.

3eme Algorithme

Conclusion



Améliorations possibles

Temps de réponse plus court

Meilleure qualité graphique

Solutions possibles

- Possible utilité des graphes
- Recherche exhaustive avec des permutations

FIN

MERCI DE NOUS AVOIR ÉCOUTÉES

LEVACHER Ethan

DESPERROIS Lucas

COLLIN Ethan