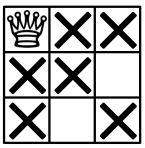
DS 4 d'informatique - MPSI - Correction

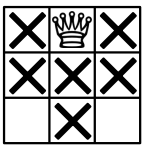
Question 1.a – On distingue 3 cas :

→ Si la première dame est placée au centre de l'échiquier, elle attaque toutes les autres cases et on ne peut pas placer la deuxième dame :

→ Si la première dame est placée dans un coin, il reste deux cases non attaquées. Étant donné que ces deux cases sont sur la même diagonale, il est impossible de placer les deux autres dames :



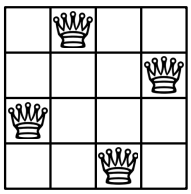
→ Si la première dame est placée sur un bord, il reste deux cases non attaquées. Étant donné que ces deux cases sont sur la même ligne ou la même colonne, il est impossible de placer les deux autres dames :

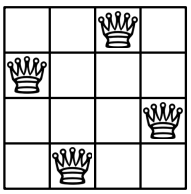


En conclusion :

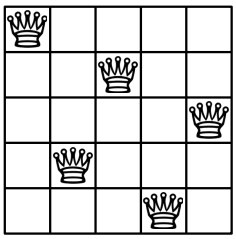
Il n'y a pas de solution au problème des n dames pour $n = 3$.

Question 1.b – Pour $n = 4$, il y a deux solutions au problème des n dames :





Question 1.c – Voici une solution possible :



Question 2 –

```
def verifElem(posDames):
    n = len(posDames)
    for e in posDames:
        if e < 0 or e >= n:
            return False
    return True
```

Question 3.a –

```
1  def makeNbParCol(posDames):
2      n = len(posDames)
3      nbParCol = [0]*n
4      for i in range(n):
5          j = posDames[i]
6          nbParCol[j] += 1
7      return nbParCol
```

Question 3.b –

```
1  def verifCol(posDames):
2      nbParCol = makeNbParCol(posDames)
3      for e in nbParCol:
4          if e != 1:
5              return False
6      return True
```

Question 3.c –

⋆ La fonction `makeNbParCol` est de complexité linéaire en n. En effet, la création de la ligne 7 s'exécute en temps linéaire en n. De plus il y a n tours de boucle et chaque tour s'exécute en temps constant.

⋆ La fonction `verifCol` est de complexité linéaire en n. En effet, l'appel à la fonction `makeNbParCol` de la liste ligne 6 s'exécute en temps linéaire en n. De plus il y a au plus n tours de boucle et chaque tour s'exécute en temps constant.

Question 4 –

```
1  # On utilise un dictionnaire (d: dict[int, NoneType]) pour vérifier si tous les
2  # (i+j) sont différents
3  def verifAntiDiag(posDames):
4      d = {}
5      n = len(posDames)
6      for i in range(n):
7          j = posDames[i]
8          if i + j in d:
9              return False
10         else:
11             d[i+j] = None
12     return True
```

Question 5.a – Pour chaque case de coordonnées (i, j) dans la grille, calculons $i - j$. Par exemple, pour $n = 8$, on obtient :

0	-1	-2	-3	-4	-5	-6	-7
1	0	-1	-2	-3	-4	-5	-6
2	1	0	-1	-2	-3	-4	-5
3	2	1	0	-1	-2	-3	-4
4	3	2	1	0	-1	-2	-3
5	4	3	2	1	0	-1	-2
6	5	4	3	2	1	0	-1
7	6	5	4	3	2	1	0

On remarque alors que la case de coordonnées (i_1, j_1) et la case de coordonnées (i_2, j_2) sont sur la même diagonale si et seulement si $i_1 - j_1 = i_2 - j_2$.

Question 5.b –

```
# On applique la méthode vue en cours qui permet de traduire une fonction
# itérative en fonction récursive.
def verifDiagAux(posDames, d, i):
    if i == len(posDames):
        return True
    j = posDames[i]
    if i-j in d:
        return False
    else:
        d[i-j] = None
        return verifDiagAux(posDames, d, i+1)
```

```
def verifDiag(posDames):
    return verifDiagAux(posDames, {}, 0)
```

Question 6.a –

```
def estSol(posDames):
    b1 = verifCol(posDames)
    b2 = verifAntiDiag(posDames)
    b3 = verifDiag(posDames)
    return b1 and b2 and b3
```

Question 6.b – D'après les questions précédentes, l'appel à chacune des trois fonctions intermédiaires s'exécute en temps linéaire en n. La fonction `estSol` s'exécute donc en temps linéaire en n.

Question 7 –

```
def configAlea(n):
    posDames = [i for i in range(n)]
    for i in range(n-1, 0, -1):
        j = random.randint(0,i)
        tmp = posDames[i]
        posDames[i] = posDames[j]
        posDames[j] = tmp
    return posDames
```

Question 8 –

```
def solAlea(n):
    assert n == 0 or n == 1 or n >= 4
    while True:
        posDames = configAlea(n)
        if estSol(posDames):
            return posDames
```

Question 9.a –

```
# Suppose qu'au moins une dame est attaquée.
def dameABouger(posDames, nbAtt):
    n = len(posDames)
    while True:
        i = random.randint(0,n-1)
        j = posDames[i]
        if nbAtt[i][j] != 1:
            return i
```

Question 9.b –

```
def nvPos(i, nbAtt):
    L = nbAtt[i]
    n = len(L)
    if L[0] <= L[1]:
        j1 = 0; j2 = 1
    else:
        j1 = 1; j2 = 0
    for j in range(2,n):
        if L[j] < L[j1]:
            j2 = j1; j1 = j
        elif L[j] < L[j2]:
            j2 = j
    return j1, j2
```

Question 9.c –

```
# Attention: modifie la liste donnée en entrée
def etape2b(posDames, nbAtt):
    i = dameABouger(posDames, nbAtt)
    j1,j2 = nvPos(i, nbAtt)
    if j1 == posDames[i]:
        posDames[i] = j2
    else:
        posDames[i] = j1
    return posDames
```

Question 10 –

```
def makeNbAtt(posDames):
    n = len(posDames)
    attLignes = {i: 0 for i in range(n)}
    attCol = {j: 0 for j in range(n)}
    attAntiDiag = {k: 0 for k in range(2*n-1)}
    attDiag = {k: 0 for k in range(-n+1, n)}
    for i in range(n):
        j = posDames[i]
        attLignes[i] += 1
        attCol[j] += 1
        attAntiDiag[i+j] += 1
        attDiag[i-j] += 1
    nbAtt = [[attLignes[i] + attCol[j] + attAntiDiag[i+j] + attDiag[i-j]
              for j in range(n)] for i in range(n)]
    ### La dame (i,j) n'attaque qu'une fois la case (i,j)
    for i in range(n):
        nbAtt[i][posDames[i]] -= 3
    return nbAtt
```

Question 11 –

```
def solRI(n):
    while True:
        posDames = configAlea(n)
        # print("restart")
        if estSol(posDames):
            return posDames
        for _ in range(1000):
            nbAtt = makeNbAtt(posDames)
            posDames = etape2b(posDames, nbAtt)
            if estSol(posDames):
                return posDames
```

Question 12 –

```
# Indique si la case de coordonnées (i,j) est dans la grille de taille n x n
def estDansGrille(i,j,n):
    return 0 <= i and i <= n-1 and 0 <= j  and j <= n-1
```

```python
# Cette fonction renvoie une liste de booléens att telle que att[j] vaut True
# lorsque la case de coordonnées (i,j) est attaquée par une dame d'indice
# i0 < i.
def makeAtt(i, posDames):
    n = len(posDames)
    att = [False] * n
    for i0 in range(i):
        j0 = posDames[i0]
        ### Attaque sur la colonne
        att[j0] = True
        ### Attaque sur l'anti-diagonale
        j = i0 + j0 - i
        if estDansGrille(i,j,n):
            att[j] = True
        ### Attaque sur la diagonale
        j = i - i0 + j0
        if estDansGrille(i,j,n):
            att[j] = True
    return att
```

```python
# On suppose que les i premières dames ont été placées.
# Cette fonction place la dame numéro i puis place les autres dames avec des
# appels récursifs.
def nbSolAux(i, posDames):
    n = len(posDames)
    if i >= n:
        return 1
    att = makeAtt(i, posDames)
    L = [j for j in range(n) if not att[j]]
    res = 0
    for j in L:
        posDames[i] = j
        res += nbSolAux(i+1, posDames)
    return res
```

```python
# Hypothèse: n >= 0.
def nbSol(n):
    posDames = [0]*n
    return nbSolAux(0, posDames)
```