Partie I – Préliminaires et vérification d'une solution

Q1.

```python
1 def cases_noires(cle_l) :
2     total = 0
3     for cles in cle_l :
4         for x in cles :
5             total += x
6     return total
```

La ligne 5 a une complexité en `O(1)`.
Le nombre maximal de blocs sur une ligne de taille `nc` est $\lfloor nc/2 \rfloor$ donc la boucle des lignes 4 et 5 a une complexité en `O(nc)`.
Avec la boucle de ligne 3 on obtient que la complexité de la fonction est en `O(nl*nc)`.

Q2.

```python
def compatible(cle_l, cle_c) :
    return cases_noires(cle_l) == cases_noires(cle_c)
```

Q3.

```python
def taille_minimale(cles) :
    taille = 0
    debut = True
    for valeur in cles :
        taille += valeur
        if not debut :
            taille += 1
        debut = False
    return taille
```

Q4. L'instruction `verif_ligne([[1,0,0,1]], [[2,1]], 0)` renvoie `False` à la ligne 13 car le premier bloc n'est pas de la bonne taille (le test `i_bloc < len(cle_l[i])` est faux). L'instruction `verif_ligne([[1,0,1,0]], [[1]], 0)` renvoie `False` à la même ligne car le nombre de blocs n'est pas le bon (le test `taille == cle_l[i][i_bloc]` est faux).
La fonction `verif_ligne` ne vérifie pas que la ligne comporte le bon nombre de blocs.
Par exemple `print(verif_ligne([[1,0,0,0,1]], [[1,1,2]], 0))` renvoie `True`.

Partie II – Résolution systématique

Q5. On a $n = k * nc + l$ donc $k = n//nc$ et $l = n\%nc$.

Q6.

```
1 def liste_solutions(cle_l, cle_c) :
2     nl = len(cle_l)
3     nc = len(cle_c)
4     sol_p = init_sol(nl, nc, -1)
5     liste=[]
6     n = 0
7     ################################
8     def liste_solutions_aux(n ,sol_p, liste) :
9         if n == nc*nl : # cas de base
10            if verif(sol_p, cle_l, cle_c) :
11                liste.append(sol_p)
12        else :
13            k = n // nc
14            l = n % nc
15            # on teste s'il y a une solution où la case n+1 est blanche
16            sol_p_c = copy_sol(sol_p)
17            sol_p_c[k][l] = 0
18            liste_solutions_aux(n+1, sol_p_c, liste)   # appel récursif
19            # on teste ensuite s'il y a une solution où la case n+1 est noire
20            sol_p_c = copy_sol(sol_p)
21            sol_p_c[k][l] = 1
22            liste_solutions_aux(n+1, sol_p_c, liste)   # appel récursif
23    #####################################
24    liste_solutions_aux(n, sol_p, liste)
25    return liste
```

La fonction génère toutes les grilles dont `2**(nl)*(nc)` grilles et leur applique `verif` qui est en `O(nl*nc)` donc la complexité est en `O(nl*nc*2**(nl*nc))` ce qui est très mauvais.

Q7. On peut vérifier que les `k` premières lignes ne contiennent pas trop de cases noires car dans ce cas il est inutile de lancer les appels récursifs.
Entre les lignes 14 et 15 on ajoute

```
            # on teste si les k premières lignes ne contiennent pas trop de cases noires
            test = True
            for i in range(k) :
                if not verif_ligne(sol_p, cle_l, i) :
                    test = False
            # on ne lance les appels récursifs que si le test est passé
            if test :
```

Partie III - Placements possibles d'un bloc

Q8.

```
1 def conflit(c, s) :
2     if c!=0 and sol_p[i_ligne][c-1] == 1 :
3         return c-1
4     for i in range(c,c+s) :
5         if sol_p[i_ligne][i] == 0 :
6             return i
7     if c + s < nc and sol_p[i_ligne][c+s] == 1 :
8         return c+s
9     return nc
```

Les instructions sont toutes en `O(1)` donc avec la boucle de la ligne 4 la complexité de cette fonction est en `O(s)`.

Q9.

```python
1  def prochain(c, s) :
2      s_temp = 0
3      for i in range(c,nc) :
4          if sol_p[i_ligne][i] == 0 :
5              # on ne peut pas mettre de bloc
6              s_temp = 0
7          elif sol_p[i_ligne][i] == -1 :
8              s_temp += 1
9          else :
10             s_temp += 1
11         if s_temp == s :
12             return i-s+1
13     return -1
```

Dans la boucle de la ligne 3 les instructions sont en `O(1)` donc la complexité de cette fonction est en `O(nc)`.

Partie IV - Placements possibles de tous les blocs d'une ligne

Q10.

```
1 def calcul_matrice(M) :
2     B = len(cle_l[i_ligne])
3     for b in range(1, B):
4         s = cle_l[i_ligne][b]
5         for c in range(1,nc):
6             if c>0 and M[c-1][b] >= 0 and sol_p[i_ligne][c] != 1 :
7                 M[c][b] = M[c-1][b]
8             elif 0 <= c-s-1 and conflit(c-s+1, s) > c and M[c-s-1][b-1] >= 0 :
9                 M[c][b] = c-s+1
10            else :
11                M[c][b] = -1
```

Les lignes 6, 7, 9 et 11 ont une complexité en `O(1)`. La ligne 8 a une complexité en `Os`.
Avec la boucle de la ligne 5 on a donc une complexité en `O(nc*s)`.
Lignes 3 et 4 la variable `s` prend comme valeur la taille des blocs, et la somme de la taille de ces blocs est majorée par `nc`, donc la complexité totale de la fonction est en `O(nc**2)`.

Q11. La première case noire doit faire partie du premier bloc :

- ▷ s'il n'y en a pas on peut placer le bloc en position `0`;
- ▷ s'il y en a une en position p le bloc devrait commencer en positions `max(0,p-s+1)`

```
if c < p :  # pas de case noire dans les c premières cases
    if c==0 :
        if s == 0 and sol_p[i_ligne][0] == -1 :
            M[c][0] = 0
        else :
            M[c][0] = -1
    else :
        if M[c-1][0] >= 0 : # on avait placé le bloc entre les cases 0 et c-1
                            # et la case c n'est pas noire
            M[c][0] = M[c-1][0] # on place le bloc au même endroit
        else :
            if sol_p[i_ligne][0] != 0 and 0 <= c-s+1 and conflit(c-s+1, s) > c :
                # on n'avait pas réussi à le placer entre 0 et c-1
                # et la case c n'est pas blanche
                M[c][0] = c-s+1  # on place le bloc pour qu'il se termine en c si c'est possible
            else: # dans tous les autres cas, notamment quand la case c est blanche
                M[c][0] = -1 # on ne peut pas le placer
elif c == p : # la case c est la première case noire
    if 0 <= c-s+1 and conflit(c-s+1, s) > c : # si on peut placer le bloc de c-s+1 à c
        M[c][0] = c-s+1  # on place le bloc pour qu'il se termine en c
    else :
        M[c][0] = -1
elif c >p : # la première case noire a déjà été rencontrée
    if M[c-1][0] >= 0 : # on a pu placer le bloc entre 0 et c-1
        M[c][0] = M[c-1][0]    # o garde la même place
    else: # on n'a pas pu placer le premier bloc entre 0 et c-1
        if sol_p[i_ligne][0] == -1 : # la case c est indéterminée
            if 0 <= c-s+1 and conflit(c-s+1, s) > c :
                M[c][0] = c-s+1  # on place le bloc pour qu'il se termine en c
        elif sol_p[i_ligne][0] == 1 : # la case c est noire
            if p >= c-s+1 and 0 <= c-s+1 and conflit(c-s+1, s) > c :
                # les cases p et c peuvent être recouvertes par le premier bloc
                M[c][0] = c-s+1  # on place le bloc pour qu'il se termine en c
        else: # dans tous les autres cas, notamment si la case c est blanche
            M[c][0] = -1
```

Q12.

```python
def premiere_case(M) :
    L = []
    B = len(cle_l[i_ligne]) # nombre de blocs
    for b in range(B-1,-1,0) : # on commence par les blocs les plus à droites
        if M[nc-1][b] != -1 :
            L.append(M[nc-1][b])
        else:
            return []
    return L
```

Q13.

```python
def remplissage(liste_pp, liste_dp) :
    B = len(liste_pp)  # nombre de blocs
    for b in  range(B) :  # on parcourt les blocs
        s = cle_l[i_ligne][b] # taille du bloc b
        fin_min = liste_pp[b] + s - 1 # valeur minimale de la position
                                      # de la dernière case du bloc b
        debut_max = liste_pp[b] # valeur maximale de la position de la première case
        if fin_min >= debut_max :
            for j in range(debut_max, fin_min+1) :
                sol_p[i_ligne][j] = 1  # on modifie sol_p par effet de bord
```

Q14.

```python
def cases_blanches(liste_pp, liste_dp) :
    B = len(cle_l[i_ligne]) # nombre de blocs
    for i in range(nc) :
        if sol_p[i_ligne][i] == - 1 : # case indéterminée
            # on recherche la case blanche la plus à gauche
            blanc_g = i - 1
            while blanc_g >= 0 and sol_p[i_ligne][blanc_g] != 0 :
                blanc_g -= 1
            # on recherche la case blanche la plus à droite
            blanc_d = i + 1
            while blanc_d < nc and sol_p[i_ligne][blanc_d] != 0 :
                blanc_d += 1
            if blanc_g != -1 and blanc_d != nc : # si ces deux cases existent
                m = blanc_d - blanc_g - 1 # taille maximale d'un bloc couvrant a case i
                test = False # on cherche si m est compatible avec la taille des blocs
                             # qui conviendraient
                for b in range(B) :
                    if liste_pp[b] <= i and liste_dp[b] >= i :
                    # le bloc b pourrait contenir la case i
                        if cle_l[i_ligne][b] <= m : # si sa taille convient
                            test = True
                if not test : # on a trouvé aucun bloc qui convient
                    sol_p[i_ligne][i] = 0 # la case i est blanche
                    # on modifie sol_p par effet de bord
```