Corrigé

Concours blanc info

Exercice 1.

1.

```
1  def cases_noires(cle_l) :
2      res=0
3      for cle in cle_l : # parcours des clés
4          for v in cle : # nb de cases noires par clé
5              res+=v
6      return res
```

La complexité est $O(nl \times nc)$ car la première boucle for a au plus nl clés et la seconde au plus nc valeurs. L'opération dans la boucle étant en $O(1)$.

2.

```
1  def compatibles(cle_l,cle_c) :
2      return cases_noires(cle_l)==cases_noires(cle_c)
```

3.

```
1  def taille_minimale(l) :
2      n=len(l)
3      res=n-1 # nombre de cases blanches minimales
4      for v in l : # nb de cases noires de la liste
5          res+=v
6      return res
```

4. 1. C'est le test de la ligne 9 qu'il faut étudier. Contre-exemples donnés avec $nl = 1$ et $nc = 2$.
   - False pour premier test : `sol=[[1,1]]` et `cle_l=[[]]` et `i=0` car `i_bloc>=len(cle_l[i])` puisque `len(cle_l[i])=0`.
   - False pour deuxième test : `sol=[[1,1]]` et `cle_l=[[1]]` et `i=0` car `taille>cle_l[i][i_bloc]`.

   2. résultat incorrect : `sol=[[0,0]]` et `cle_l=[[1]]` et `i=0`.
   En effet le programme renvoie True dans le cas où il y a des blocs supplémentaires dans `cle_l` à la suite des blocs corrects.
   On peut aussi donner un exemple de résultat incorrect si l'on trouve autre chose que 0 ou 1 dans `sol`.
   Une modification possible : on remplace la dernière ligne par :
   `return i_bloc==len(cle_l[i])`

5. $n = k * nc + l$ donc l est le reste de la division euclidienne de n par nc et k le quotient.

6.

```
1  def listes_solutions(cle_l,cle_c) :
2      sol_p=init_sol(nl,nc,-1)
3      def liste_solutions_aux(n,sol_p,liste) :
4          if n==nc*nl :
5              if verif(sol_p,cle_l,cle_c) :
6                  sol_p_copie=copy_sol(sol_p)
7                  liste.append(sol_p_copie)
8          else :
9              k=n//nc
10             l=n%nc
11             sol_p[k][l]=0
12             liste_solutions_aux(n+1,sol_p,liste)
13             sol_p[k][l]=1
14             liste_solutions_aux(n+1,sol_p,liste)
15     liste=[]
16     liste_solutions_aux(0,sol_p,liste)
17     return liste
```

La complexité est en $O\left(nc \times nl \times 2^{nl \times nc}\right)$ car il y a $2^{nc \times nl}$ matrices à tester, chaque test étant de complexité $O(nl \times nc)$.

7. Une façon de procéder
   - Entre les lignes 1 et 2 on insère un code permettant de calculer le nombre de cases noires attendues dans chaque colonne et chaque ligne à l'aide de `cle_l` et `cle_c`.

```
    tab_l=[cases_noires([cle_l[i]]) for i in range(nl)]
    tab_c=[cases_noires([cle_c[j]]) for j in range(nc)]
```

À la place de la ligne 14 on insère le code suivant, qui ne lance `liste_solutions_aux` que si cela vaut le coup en testant si l'on a pas dépassé le nombre de cases noires possible :

```
            noire_l=0
            for j in range(l+1) :
                noire_l=noire_l+sol_p[k][j]
            noire_c=0
            for i in range(k+1) :
                noire_c=noire_c+sol_p[i][l]
            if noire_l<=tab_l[k] and noire_c<=tab_c[l] :
                liste_solutions_aux(n+1,sol_p,liste)
```

8.

```
def conflit(c,s) :
    if c>0 and sol_p[i_ligne][c-1]==1 :
        return c-1
    for j in range(s) :
        if sol_p[i_ligne][c+j]==0 :
            return c+j
    if c+s<nc and sol_p[i_ligne][c+s]==1 :
        return c+s
    return nc
```

La complexité est en $O(s)$ par lecture directe de la boucle for.

9.

```
def prochain(c,s) :
    j=c
    while j+s<=nc : # tant qu'il y a assez de place à droite
        while j>0 and sol_p[i_ligne][j-1]==1 : # tant qu'il y a des cases noires
            j+=1                               # à gauche, on décale le pointeur
        if j+s>nc :   # si plus de place à droite
            return -1 # échec
        k=0 # calcul de la place disponible pour le bloc
        while k<s and sol_p[i_ligne][j+k]!=0 :
            k+=1
        if k==s : # si on a trouvé s cases non blanches
            if j+k==nc : # si on est en bout de ligne
                return j # gagné
            if sol_p[i_ligne][j+k]!=1 : # si la case suivante n'est pas noire
                return j                # gagné
        j=j+k+1 # sinon on se place après la dernière case noire rencontrée
    return -1 # échec à la fin de toutes les tentatives
```

La complexité est en $O(nc)$ car c'est le nombre maximum d'incrémentations de j et de k cumulées, et pour chaque incrémentation il y a au plus 4 opérations en temps constant.

10. On traduit directement les instructions de l'énoncé :

```
def calcul_matrice(M) :
    B=len(cle_l[i_ligne])
    for b in range(1,B) :
        s=cle_l[i_ligne][b]
        for c in range(1,nc) :
            if M[c-1][b]>=0 and sol_p[i_ligne][c]!= 1 :
                M[c][b]=M[c-1][b]
            elif c-s-1>=0 and M[c-s-1][b-1]>=0 and conflit(c-s+1,s)>c :
                M[c][b]=c-s+1
                M[c][b-1] = M[c-1][b-1]
            else :
                M[c][b]=-1
```

Attention cependant à une subtilité non déclarée dans l'énoncé, si l'on ajoute le dernier bloc en fin de ligne à cause de l'échec au test de la ligne 6, en pratique on place alors les **deux** derniers blocs (par rapport à la valeur précédente de c, on «détache» l'avant dernier bloc de la case noire et on y met le dernier à la place). C'est la raison de la ligne 10.,

La complexité est en $O(nc^2)$ car il y a une double boucle for et car $B \leqslant nc$.

11. Quelques observations générales pour «digérer» la réponse.

- 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 position $\max(0, p-s+1)$.
- Structurellement, on va distinguer les cas $c < p$, $c = p$ et $c > p$.
  — Dans le premier cas le sous-cas $c = 0$ est spécial ;
  — dans le second on pourra l'intégrer au cas «général» ;
  — Attention, dans le troisième cas la case d'indice p fait partie du bloc mais n'est pas nécessairement la première du bloc.

Un programme commenté possible est le suivant :

```
if c < p : # pas de case noire dans les c premières cases
    if c==0 :
        if s == 0 and sol_p[i_ligne][c] == -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][c] != 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
    else : # 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][c] == -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][c] == 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
```

12.

```
def premiere_case(M) :
    res=[]
    for b in range(len(cle_l[i_ligne])) :
        v=M[nc-1][b]
        if v<0 :
            return []
        else :
            res.append(v)
    return res
```

13.

```
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_dp[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
```

14.

```
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
```