XENS ITC 2024 - Correction

Q1 On doit sommer tous les entiers apparaissant dans les listes de `cle_l` :

```
def cases_noires(cle_l):
    s = 0
    for l in cle_l:
        for n in l:
            s = s + n
    return s
```

La complexité est $O\left(\sum_{i=0}^{\texttt{len(cle\_l)}} \texttt{cle\_l}[i]\right)$.

Q2 La fonction `cases_noires` appliquée à `cle_c` renvoie aussi le nombre de cases noires de la solutions.

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

Q3 Si `L` est la liste d'entiers, on aura besoin de `len(L) - 1` cases blanches au minimum en plus des cases noires.

```
def taille_minimale(L):
    s = len(L) - 1
    for i in L:
        s = s + i
    return s
```

Q4

- En prenant `sol = [[1]]`, `cle_l = [[10]]`, la fonction renvoie `False` car le test `taille == cle_l[0][0]` échoue.
  En prenant `sol = [[1, 0, 1]]`, `cle_l = [[1]]`, la fonction renvoie `False` car le test `i_bloc == len(cle_l[i])` échoue à la dernière itération.
- En prenant `sol = [[0]]`, `cle_l = [[10]]`, la fonction renvoie `True` alors que la solution n'est pas bonne. Plus généralement, s'il n'y a que des cases blanches dans la solution, la fonction renvoie `True`. Après la boucle, il faut encore vérifier que `i_bloc == len(cle_l)`.

Q5 On a donc $n = k \times nc + \ell$. Comme $\ell < nc$, k et ℓ sont le quotient et le reste de la division euclidienne de n par nc.

Q6 C'est un algorithme de backtracking.

```
def liste_solutions(cle_l, cle_c):
    def liste_solutions_aux(n, sol_p, liste):
        if n == nc*nl and verif(sol_p, cle_l, cle_c):
            liste.append(copy_sol(sol_p))
        else:
            k, l = n//nc, n\%nc
            sol_p[k][l] = 0
            liste_solutions_aux(n+1, sol_p, liste)
            sol_p[k][l] = 1
            liste_solutions_aux(n+1, sol_p, liste)
            sol_p[k][l] = -1
    liste = []
    sol_p = init_sol(nl, nc, -1)
    liste_solutions_aux(0, sol_p, liste)
    return liste
```

Si on note C_n le coût de l'appel de `liste_solutions_aux` avec n, on a $C_n = 2C_{n+1}$ et $C_{\texttt{nc*nl}} = O(\texttt{nc*nl})$, donc la complexité est $O(2^{\texttt{nc*nl}}\texttt{*nc*nl})$.

Q7 Avant de définir la fonction auxiliaire, on commence par définir deux listes contenant le nombre de cases noires dans chaque ligne et dans chaque colonne de la solution. On rajoute deux paramètres à la fonction auxiliaire : `nb` et `col`. Le premier est un entier qui compte combien on a mis de cases noires dans la ligne courante et le second est une liste d'entiers qui compte combien on a mis de cases noires jusqu'à présent dans chaque colonne. Avant de mettre un 1 dans la case `k, l` on vérifie si on ne dépasse pas le nombre maximal de cases noires dans la ligne/colonne correspondante.

Q8 Par exemple :

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

Chaque accès et test se fait en $O(1)$ et on en fait $s+2$.

Q9 Par exemple :

```
def prochain(c, s):
    i = c
    if c-1 >= 0 and sol_p[i_ligne][c-1] == 1:
        while i < nc and sol_p[i_ligne][i] == 1:
            i = i + 1
        i = i + 1
    l = 0
    while l < s and i + l < nc:
        if sol_p[i_ligne][i + l] == 0:
            i = i + 1
            l = 0
        else:
            l = l + 1
        if l == s and i+l < nc and sol_p[i_ligne][i+l] == 1:
            while i+l < nc and sol_p[i_ligne][i+l] == 1:
                l = l + 1
            if sol_p[i_ligne][i+l-s] != 1:
                return i+l-s+1
            else:
                i = i+l
                l = 0
    if l == s:
        return i
    return -1
```

On ne parcourt qu'une seule fois toutes les cases de la ligne `i_ligne`, donc la complexité est bien $O(\texttt{nc})$.

Q10 Par exemple :

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

Q11 La première case noire ne peut pas être avant le premier bloc.
Déjà, `M[c][0] = -1` pour `c < s-1`. Sinon, pour calculer `M[c][0]` :

- si `M[c-1][0] >= 0` et `sol_p[i_ligne][c] != 1` alors `M[c][0] = M[c-1][0]` ;
- sinon, on vérifie :
  - ▷ `conflit(c-s+1, s) > c` ;

▷ `p >= c-s+1`

et alors `M[c][0] = c-s+1` ;

- sinon, `M[c][0] = -1`