Logimage

I Préliminaires et vérification d'une solution

Q1 Le nombre de cases noires d'une solution est la somme du nombre de cases noires par ligne, càd. la somme des clés de cette ligne.

```
def cases_noires(cle_l):
    nb_cases_noires = 0
    for i in range(len(cle_l)):
        for j in range(len(cle_l[i])):
            nb_cases_noires += cle_l[i][j]
    return nb_cases_noires
```

Complexité de la fonction `cases_noires` :

La complexité de la fonction dépend de `nl = len(cle_l)` et du nombre de clés par ligne, majoré par `nc`, le nombre de colonnes de la grille.

- À l'extérieur des boucles **for** imbriquées, les instructions sont en $\mathcal{O}(1)$.
- Le nombre de tours des boucles **for** imbriquées est égal au nombre de clés *nb_clés*.
  À chacun de ces *nb_clés* tours de boucle, l'instruction exécutée est en $\mathcal{O}(1)$.
  La complexité des boucles **for** imbriquées est donc en $\mathcal{O}(nb\_clés)$.

Par somme, la complexité de la fonction est en $\mathcal{O}(nb\_clés)$.
Or, le nombre de clés est au plus égal à $nl \times \lceil \frac{nc}{2} \rceil = nl \times \mathcal{O}(nc)$.

La complexité de la fonction `cases_noires` est donc en $\boxed{\mathcal{O}(nl \times nc)}$ où nl est le nombre de lignes et nc le nombre de colonnes de la grille.

Q2 Le nombre de cases noirs par le décompte par ligne doit être égal au nombre de cases noirs par le décompte par colonne.

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

Q3 La taille minimale d'une ligne est obtenue en mettant un bloc de cases noires (éventuellement le même bloc) à chaque extrémité de la ligne et une seule case blanche entre chaque bloc. La taille minimale d'une ligne est alors égale au nombre de cases noires auquel on ajoute le nombre de clés moins 1. On note que cette formule s'applique bien lorsque l'on a 1 seule clé. Ceci permet d'écrire la fonction `taille_minimale` suivante :

```
def taille_minimale(t):
    '''t est une liste non vide d'entiers.
    Postcondition: la taille minimale est un entier naturel.'''
    nb_cases = 0
    for i in range(len(t)):
        nb_cases += t[i]
    return nb_cases + len(t) - 1
```

Q4 **1.** Exemples d'arguments `sol` et `cle_l` pour lesquels `verif_ligne` renvoie `False`.

- Cas 1 : 2 blocs de cases noires et `cle_l` contient une sous-liste avec une seule clé.
  $\boxed{\texttt{sol = [[1, 0, 1]] ; cle\_l = [[1]] ; i = 0}}$.

  Explication :
  Ici, `nl` vaut 1 et `nc` vaut 3.
  Lorsque `j` vaut 2, `taille` prend la valeur 1 et `j+1 == nc` (la condition du **if** de la ligne 8 est vérifiée). Comme `i_bloc` vaut 1 (car on avait 1 bloc d'une case noire quand `j` valait 1) et `len(cle_l[i])` vaut 1, on a `i_bloc >= len(cle_l[i])` (la condition du **if** de la ligne 9 n'est pas vérifiée). Donc, on renvoie `False`.

— Cas 2 : 1 bloc avec 1 case noire et `cle_l` contient une sous-liste avec une clé indiquant 1 bloc de 2 cases noires.
`sol = [[1, 0]] ; cle_l = [[2]] ; i = 0`.
Explication :
Ici, `nl` vaut 1 et `nc` vaut 2.
Lorsque `j` vaut 0, `couleur_case` vaut 1, donc `taille` prend la valeur 1.
Lorsque `j` vaut 1, `couleur_case` vaut 0, donc `taille` garde la valeur 1.
Ainsi `taille > 0` et `couleur_case == 0` (la condition du `if` de la ligne 8 est vérifiée).
Comme `i_bloc` vaut 0 et `len(cle_l[i])` vaut 1, on a `i_bloc < len(cle_l[i])`.
Mais `cle_l[i][i_bloc]` vaut 2, donc `taille != cle_l[i][i_bloc]` (la condition du `if` de la ligne 9 n'est pas vérifiée). Donc, on renvoie `False`.

2. La fonction proposée ne vérifie pas les points suivants :

— Elle ne vérifie **pas** que la ligne d'indice `i` ne contient que des 0 et des 1.
Par exemple, `sol = [[1, -1, 1]] ; cle_l = [[2]] ; i = 0` renvoie `True`.

On peut modifier la fonction en ajoutant entre les lignes 5 et 6 de l'énoncé les instructions :

```python
if couleur_case != 0 and couleur_case != 1:
    return False
```

— Elle ne vérifie **pas** que le nombre de blocs est égal aux nombres de clés de `cle_l`.
Par exemple, `sol = [[0, 0, 1]] ; cle_l = [[1, 1]] ; i = 0` renvoie `True`.
En effet, le bloc de taille 1 est en phase avec `cle_l[0][0]`, mais le nombre total de blocs est incohérent.

De même, et pour les mêmes raisons, `sol = [[0, 0, 0]] ; cle_l = [[1]] ; i = 0` renvoie `True`.

On peut modifier la fonction en remplaçant la ligne 14 par l'instruction :

```python
return i_bloc == len(cle_l[i])
```

II Résolution systématique

Q5 Par définition, $n = k \times nc + l$. Ainsi, `k = n // nc` et `l = n % nc`.

Q6 On utilise la méthode proposée par l'énoncé :

```python
def liste_solution(cle_l, cle_c):
    sol_p = init_sol(nl, nc, -1)
    n = 1 # notation de l'énoncé
    liste = []
    liste_solutions_aux(n-1, sol_p, liste) # n-1 afin de démarrer à l'indice (0,0) de sol_p
    return liste

def liste_solutions_aux(n, sol_p, liste):
    if n == nl * nc:
        if verif(sol_p, cle_l, cle_c):
            sol_p_c = copy_sol(sol_p) # car sol_p est modifiée par la fonction appelante
            liste.append(sol_p_c)
    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)
```

Complexité de la fonction `liste_solutions` :
La fonction `liste_solutions` génère $2^{nl \times nc}$ solutions.
Chaque solution est vérifiée par la fonction `verif` de complexité en $\mathcal{O}(nl \times np)$.
La complexité de `liste_solutions` est donc en $\boxed{\mathcal{O}(nl \times np \times 2^{nl \times nc})}$.
La complexité de `liste_solutions` est donc **exponentielle**.

Q7 On peut considérer deux tables, l'une avec le nombre de cases noires par ligne, et l'autre avec le nombre de cases noires par colonne. Ces tables sont alimentées par la fonction `liste_solution` à partir respectivement de `cle_l` et `cle_c` avant l'appel de la fonction `liste_solutions_aux`.

Dans le code de la fonction `liste_solutions_aux`, juste après avoir mis à jour `sol_p` avec 1, on peut vérifier si le total des 1 de la ligne `k`, pour les indices colonnes allant de 1 à `l`, reste inférieur au total de 1 sur cette ligne. Si tel n'est pas le cas, on n'appelle pas `liste_solutions_aux(n+1, sol_p, liste)`. On procède de même si le total des 1 de la colonne `l`, pour les indices lignes allant de 1 à `k`, reste inférieur au total de 1 sur cette colonne.

III Placement possibles d'un bloc

Q8 On développe une fonction `mini3` dont l'objectif est de trouver le minimum de 3 entiers.

```python
def mini2(a, b):
    if a < b:
        return a
    return b

def mini3(a, b, c):
    return mini2(mini2(a, b), c)

def conflit(c, s):
    '''Préconditions: 0 <= c < nc, 1 <= s et c + s <= nc où nc est la variable globale désignant le nombre de colonnes. Les variables sol_p et i_ligne, resp. une solution provisoire et i_ligne l'une des lignes de sol_p, sont des variables gloables.'''
    ligne = sol_p[i_ligne]
    c1 = c2 = c3 = nc
    if c >= 1 and ligne[c-1] == 1: # cas a
        c1 = c-1
    for i in range(c, c+s): # cas b
        if ligne[i] == 0:
            c2 = i
    if c+s < nc and ligne[c+s] == 1: # cas c
        c3 = c+s
    return mini3(c1, c2, c3)
```

Complexité de la fonction `conflit` :

La complexité de la fonction `conflit` dépend de `s`.

- À l'extérieur de la boucle `for`, les instructions sont en $\mathcal{O}(1)$.
- Pour chacun des `s` tours de la boucle `for`, on exécute des instructions en $\mathcal{O}(1)$. La complexité de la boucle `for` est donc en $\mathcal{O}(s)$.

Par somme, la complexité de la fonction `conflit` est en $\mathcal{O}(s)$.

Q9

```python
def prochain(c, s):
    ligne = sol_p[i_ligne]
    i = c
    pos_valide = c
    while i < nc:
        # cas de conflit a
        if i >= 1 and ligne[i-1] == 1:
            if i == pos_valide: # début de bloc noir
                pos_valide = i+1
            i += 1
        # cas de conflit b ou de conflit c
        elif (i < pos_valide + s and ligne[i] == 0) or (i+s < nc and ligne[i+s] == 1):
            i += 1
            # écriture de la première position valide (on gère le cas particulier d'une ligne complète de 0)
            if i+1 < nc and ligne[i+1] != 0 and i - pos_valide >= s:
                return pos_valide
            pos_valide = i
        # cas sans conflit
        else:
            i += 1
    if pos_valide + s > nc: # bloc trop long
        return -1
    else: # écriture de la première position valide
        return pos_valide
```

Complexité de la fonction `prochain` :

La complexité de la fonction `prochain` dépend de `nc`.

- À l'extérieur de la boucle **`while`**, les instructions sont en $\mathcal{O}(1)$.
- La boucle **`while`** exécute au plus `nc` tours de boucles (quand `c` vaut 0).
  À chaque tour de boucle, on exécute des instructions en $\mathcal{O}(1)$.
  La complexité de la boucle **`while`** est donc en $\mathcal{O}(nc)$.

Par somme, la complexité de la fonction `prochain` est en $\mathcal{O}(nc)$.

IV Placements possibles de tous les blocs d'une ligne

Q10

```python
def calcul_matrice(M):
    '''Programmation dynamique ascendante'''
    B = len(cle_l[i_ligne])
    for c in range(1, nc):
        for b in range(1, B):
            if M[c-1][b] >= 0 and sol_p[i_ligne][c] != 1:
                M[c][b] = M[c-1][b]
            else:
                s = cle_l[i_ligne][b]
                if 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 # le dernier bloc est placé tout à droite
                    M[c][b-1] = M[c-1][b-1] # l'avant dernier bloc prend la dernière position déjà déterminée
                else:
                    M[c][b] = -1
```

Quelques remarques :

— L'ordre d'écriture de la condition `M[c-s-1][b-1] != -1 and conflit(c-s+1, s) > c` permet de bénéficier de l'évaluation paresseuse de Python : l'appel à la fonction `conflit` n'est **pas** réalisé quand `M[c-s-1][b-1]` vaut -1.

— Une fois le dernier bloc positionné tout à droite, il faut que le précédent bloc reste à la place déterminée pour la valeur de `c-1`, d'où la ligne `M[c][b-1] = M[c-1][b-1]`). Sans cette instruction, l'avant-dernier bloc est "superposé" au dernier bloc (voir l'exemple ci-dessous) !

Reprenons *l'exemple de l'énoncé.*
Soit `sol_p = [[-1, 0, -1, -1, 1, -1, -1, -1, -1, 1]]` et `cle_l = [[1, 2, 3]]`, avec `i_ligne = 0`.

Alors, `nc` vaut 10 et `len(cle_l[i_ligne])` vaut 3.

La matrice `M` est une matrice à 10 lignes et 3 colonnes. Pour tout `c`, on peut positionner la première case noire en 0, ce qui donne la matrice `M` précalculée suivante :
`M = [[0, -1, -1], [0, -1, -1], [0, -1, -1], [0, -1, -1], [0, -1, -1], [0, -1, -1], [0, -1, -1], [0, -1, -1], [0, -1, -1], [0, -1, -1]].`

La fonction `calcul_matrice(M)` met à jour la matrice `M`.
Le contenu de la matrice `M` après exécution de la fonction proposée est [1] :
`M = [[0, -1, -1], [0, -1, -1], [0, -1, -1], [0, 2, -1], [0, 3, -1], [0, 3, -1], [0, 3, -1], [0, 3, -1], [0, 3, 6], [0, 3, 7]].`

Complexité de la fonction `calcul_matrice` :

La complexité de la fonction `calcul_matrice` dépend de `B = len(cle_l[i_ligne])`.

— À l'extérieur des boucles **for** imbriquées, les instructions sont en $\mathcal{O}(1)$.

— La boucle **for** d'indice `c` exécute au plus $(nc - 1)$ tours de boucle.
À chaque tour de boucle, on exécute des instructions en $\mathcal{O}(1)$ sauf lors de l'appel à la fonction `conflit` qui s'exécute en $\mathcal{O}(s)$ (cf. **Q8**).
Le nombre d'appels à `conflit` est au plus égal à $\sum_{b=1}^{B-1} s_b$ où s_b désigne la valeur de `s = cle_l[i_ligne][b]` pour la valeur de b considérée. Or, $\sum_{b=1}^{B-1} s_b \leq nc$ car il ne peut pas y avoir plus de cases noires que de cases sur la ligne considérée.
La complexité de la boucle est donc en $(nc - 1) \times \mathcal{O}(nc) = \mathcal{O}(nc^2)$.

Par somme, la complexité de la fonction `calcul_matrice` est en $\boxed{\mathcal{O}(nc^2)}$.

Q11 Étudions pour une valeur de `c` donnée, la position du premier bloc, à savoir `M[c][0]`.

Soit `p` la position de la première case noire de `sol_p[i_ligne]` quand elle existe.
S'il n'y a pas de case noire, on convient que `p` vaut -1.

Tout d'abord, notons qu'une valeur **positive** de `p` contraint la valeur de `M[c][0]`.
En effet, on a toujours $\boxed{\texttt{M[c][0]} \leq \texttt{p}}$.

Plus précisément, considérons une valeur **positive** de `p`.
On note `s = cle_l[i_ligne][0]`.

- **Soit `c < p`** : toutes les cases (quand il y en a) de 0 à `c` sont blanches ou indéterminées.

  — Soit `c = 0`.
  Si `s = 1` et `sol_p[i_ligne][c] = -1`, alors `M[0][0] = 1`, sinon `M[0][0] = -1`.

1. NB : sans l'instruction `M[c][b-1] = M[c-1][b-1]`, la dernière ligne est `[0, 8, 7]`, ce qui est erroné.

— Soit `c` ≥ 1.
  - — Si `M[c-1][0]`≥ 0, alors `M[c][0] = M[c-1][0]`
    (*une case* `c` *indéterminée ou blanche ne change pas la valeur de l'indice minimal du premier bloc*).
  - — Si `M[c-1][0] = -1` et `sol_p[i_ligne][c] = 0`, alors `M[c][0] = -1`
    (*une case* `c` *blanche ne résout pas le problème de positionnement du premier bloc*).
  - — Si `M[c-1][0] = -1` et `sol_p[i_ligne][c] = -1`, alors
    si `conflit(c-s+1, s) > c`, alors `M[c][0] = c-s+1`, sinon `M[c][0] = -1`.

- **Soit `c = p`** : toutes les cases (quand il y en a) de 0 à `c-1` sont blanches ou indéterminées et la case `c` est noire.
  - — Soit `c = 0` (*pas de case avant la case noire de* ***`sol_p[i_ligne]`***).
    Si `s = 1`, alors `M[0][0] = 1`, sinon `M[0][0] = -1`.
  - — Soit `c` ≥ 1.
    - — Si `M[c-1][0]` ≥ 0, alors `M[c][0] = M[c-1][0]`.
      (*on remonte l'indice minimal du bloc de cases noires situé strictement avant la case noire en* ***`p`***).
    - — Si `M[c-1][0] = -1` , alors
      si `conflit(c-s+1, s) > c`, alors `M[c][0] = c-s+1`, sinon, `M[c][0] = -1`.
- **Soit `c > p`** : toutes les cases de 0 à `p-1` sont blanches ou indéterminées et la case `p` est noire. On n'a aucune information sur les autres cases.
  - — Si `M[c-1][0]` ≥ 0, alors `M[c-1][0]` $\leq$ p et `M[c][0] = M[c-1][0]`.
  - — Si `M[c-1][0] = -1`, alors `M[c][0] = -1`
    (*car* ***`M[c][0]`*** *ne peut pas être strictement supérieur à* ***`p`***).

Q12 La matrice `M` obtenue à l'issue des questions **Q11** et **Q12** fournit, pour une ligne `i_ligne` donnée, les premières positions possibles des blocs noirs : il suffit de lire sa dernière ligne. Si l'une des valeurs est égale à `-1`, on renvoie une liste vide, comme demandé.

```python
def premiere_case(M):
    nl = len(M)
    for i in range(len(M[nl-1])):
        if M[nl-1][i] == -1:
            return []
    return M[nl-1]
```

Q13 D'après l'énoncé, on peut dire que `liste_pp` et `liste_dp` ont la même longueur et que l'indice `b` d'une liste donnée représente le même bloc dans l'autre liste.
Pour chaque bloc de cases noires, on noircit les cases allant de la dernière case possible du bloc (appelée `dp` dans le code) jusqu'à la case obtenue lorsqu'on passe en noir sur une longueur `s` toutes les cases à partir la première case possible du bloc (`pp` dans le code).

```python
def remplissage(liste_pp, liste_dp):
    '''procédure sans return qui met à jour sol_p[i_ligne]'''
    B = len(liste_pp) # non nul et aussi égal à len(liste_dp)
    for b in range(B):
        s = cle_l[i_ligne][b]
        pp = liste_pp[b]
        dp = liste_dp[b]
        if pp + s-1 >= dp:
            for i in range(dp, pp + s):
                sol_p[i_ligne][i] = 1
```

Q14

```python
def maxi2(a, b):
    if a < b:
        return b
    return a

def cases_blanches(liste_pp, liste_dp):
    B = len(liste_pp) # non nul et aussi égal à len(liste_dp)
    ligne = sol_p[i_ligne]

    # détermination des indices des positions des cases blanches
    blanches = [-1, -1] # positions des cases blanches (avec deux fois -1 pour traiter tous les cas dans la boucle suivante)
    for i in range(nc):
        if ligne[i] == 0:
            blanches.append(i)

    # détermination de la taille maximale m de bloc pour chaque indice de la ligne
    tailles_max = [0 for _ in range(nc)] # liste des valeurs de m
    blanche_suivante = nc
    blanche_courante = blanches.pop()
    while blanche_courante != blanche_suivante:
        for i in range(blanche_courante + 1, blanche_suivante):
            m_g = i - blanche_courante
            m_d = blanche_suivante - i
            if ligne[i] != 0:
                tailles_max[i] = maxi2(m_g, m_d)
        blanche_suivante = blanche_courante
        blanche_courante = blanches.pop()

    # détermination de la taille minimale des blocs susceptibles de couvrir une case donnée
    tailles_min_blocs = [nc+1 for _ in range(nc)] # liste des valeurs des tailles minimales des blocs recouvrant chaque case
    for b in range(B):
        s = cle_l[i_ligne][b]
        for i in range(liste_pp[b], liste_dp[b] + s):
            if s < tailles_min_blocs[i]:
                tailles_min_blocs[i] = s

    # détermination des cases blanches
    for i in range(nc):
        if tailles_min_blocs[i] > tailles_max[i]:
            ligne[i] = 0
```