Logimage

Un logimage consiste en une grille de `nl` lignes et `nc` colonnes, dans laquelle il faut retrouver une image en noir et blanc à partir de **clés**. Chaque ligne (et chaque colonne) est codée par une liste de clés entières. Un entier m dans une liste de clés correspond à un **bloc** de m cases consécutives à colorier en noir. Par exemple, pour une ligne donnée, la liste de clés $[2, 4]$ signifie qu'en regardant les cases de cette ligne de gauche à droite, on trouve d'abord un certain nombre (éventuellement nul) de cases blanches, puis un bloc de 2 cases noires consécutives, suivi d'au moins une case blanche, puis un bloc de 4 cases noires, puis éventuellement des cases blanches. De même, la liste de clés pour une colonne décrit la taille des blocs rencontrés depuis le haut jusqu'au bas de la colonne. On veut s'assurer que la solution d'un logimage existe et est unique. La figure 1 présente un exemple de logimage et sa solution : les listes en regard des lignes et des colonnes sont les clés codant la solution. Par exemple, la deuxième ligne est codée par une liste de deux clés : $[3, 1]$, alors que la première ligne est codée par une liste contenant une seule clé : $[2]$.

	[1]	[5]	[4]	[2]	[4]
[2]					
[3,1]					
[4]					
[4]					
[1,1]					

(a) Exemple de logimage

	[1]	[5]	[4]	[2]	[4]
[2]		■	■		
[3,1]	■	■	■		■
[4]		■	■	■	■
[4]		■	■	■	■
[1,1]		■			■

(b) Solution

FIGURE 1: Exemple de logimage et sa solution.

Dans ce sujet, nous représenterons les clés des lignes par une liste de listes d'entiers, de même que pour les clés des colonnes. Dans la suite du sujet, `cle_l` représentera la liste des listes de clés par lignes, et `cle_c` la liste des listes de clés par colonnes. Dans l'exemple ci-dessus, on aura donc `cle_l=[[2],[3,1],[4],[4],[1,1]]`. Une **solution** `sol` sera représentée par une liste de listes, telle que `sol[i][j]` représente l'état de la case de la grille sur la ligne `i` et la colonne `j`. Un état sera codé par un entier : 0 pour une case blanche et 1 pour une case noire.

Complexité. La complexité, ou le temps d'exécution, d'une fonction F est le nombre d'opérations élémentaires (addition, multiplication, affectation, test, ...) nécessaires à l'exécution de F. Lorsque cette complexité dépend de plusieurs paramètres n et m, on dira que F a une complexité en $O(\phi(n, m))$ lorsqu'il existe trois constantes A, n_0 et m_0 telles que la complexité de F est inférieure ou égale à $A \cdot \phi(n, m)$, pour tout $n \geq n_0$ et $m \geq m_0$. Lorsqu'il est demandé de donner la complexité d'un programme, le candidat devra justifier cette dernière si elle ne se déduit pas directement de la lecture du code.

Rappels concernant le langage Python. L'utilisation de toute fonction Python sur les listes autre que celles mentionnées dans ce paragraphe est interdite. Si `a` désigne une liste en Python de longueur n :

- `len(a)` renvoie la longueur de cette liste, c'est-à-dire le nombre d'éléments qu'elle contient ; la complexité de `len` est en $O(1)$.
- `a[i]` désigne le `i`-ème élément de la liste, où l'indice `i` est compris entre `0` et `len(a) - 1` inclus ; la complexité de cette opération est en $O(1)$.
- `a.append(e)` ajoute l'élément `e` à la fin de la liste `a` ; on supposera que la complexité de cette opération est en $O(1)$.

— `a.pop()` renvoie la valeur du dernier élément de la liste `a` et l'élimine de la liste `a` ; on supposera la complexité de cette opération en $O(1)$.

— `a.copy()` fait une copie de la liste `a` ; la complexité de cette opération est en $O(n)$.

— Si `f` est une fonction, la syntaxe `[f(x) for x in a]` permet de créer une nouvelle liste, ayant le même contenu que la liste `b` résultant de l'exécution du code suivant (et avec la même complexité) :

```
b = []
for x in a:
    b.append(f(x))
```

Manipulation de solutions. On pourra utiliser dans la suite les fonctions suivantes pour manipuler les solutions :

— `init_sol(nl,nc,v)` renvoie une nouvelle solution de `nl` lignes et `nc` colonnes dont toutes les cases sont initialisées à `v`.

— `copy_sol(sol)` renvoie une copie de la solution sol obtenue en créant une nouvelle solution de même taille et en copiant le contenu de chaque case.

La complexité de chacune de ces fonctions est en $O(\texttt{nl} \times \texttt{nc})$.

Convention. Dans le sujet, on supposera que `nl` et `nc` (qui décrivent le nombre de lignes et le nombre de colonnes de la grille) sont des variables globales accessibles à l'intérieur de toutes les fonctions.

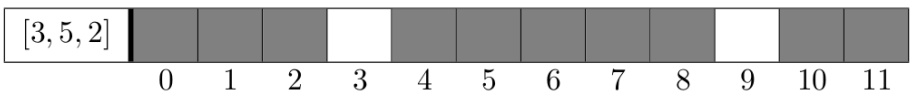
Organisation du sujet. Ce sujet comporte quatre parties, de difficulté croissante. La partie IV dépend des résultats de la partie III, autrement les parties sont indépendantes.

Partie I - Préliminaires et vérification d'une solution

Question 1. *Écrire une fonction* `cases_noires` *qui prend en argument une liste de listes d'entiers* `cle_l` *représentant les clés des lignes d'un logimage et qui renvoie le nombre de cases à colorier en noir dans la solution. Quelle est la complexité de cette fonction ?*

Question 2. *À partir de* `cle_l`*, on peut obtenir le nombre de cases noires de la solution, et de même pour* `cle_c`*. Écrire une fonction* `compatibles` *qui prend ces deux listes de listes d'entiers en arguments et qui renvoie* `True` *si et seulement si elles codent le même nombre de cases noires.*

Les règles de construction des logimages imposent au moins une case blanche entre deux blocs dans une solution valide. La clé d'une ligne permet de calculer la taille minimale de la partie de la ligne occupée par les blocs codés. Par exemple, la clé de ligne `[3,5,2]` nécessite une taille minimale de 12 cases :



Question 3. *Écrire la fonction* `taille_minimale` *qui prend en argument une liste (supposée non vide) d'entiers représentant les clés d'une ligne et renvoie cette taille minimale.*

On souhaite maintenant vérifier si une solution est valide. Pour ceci, on considère la fonction `verif_ligne` ci-dessous, qui a pour but de vérifier que la ligne d'indice `i` de la solution `sol` ne contient que des 0 et des 1 et contient tous les blocs décrits par `cle_l`. On peut s'apercevoir que cette implémentation contient une erreur, qui sera corrigée à la question suivante.

```
1  def verif_ligne(sol, cle_l, i):
2      i_bloc=0
3      taille=0
4      for j in range(nc):
5          couleur_case=sol[i][j]
6          if couleur_case==1:
7              taille=taille+1
8          if taille>0 and (couleur_case==0 or j+1==nc):
9              if i_bloc<len(cle_l[i]) and taille==cle_l[i][i_bloc]:
10                 taille=0
11                 i_bloc=i_bloc+1
12             else:
13                 return False
14     return True
```

On rappelle que l'expression `for j in range(nc)` permet d'itérer pour des valeurs de `j` allant de 0 à `nc` -1.

Question 4. *Pour la fonction* `verif_ligne` *ci-dessus, donner des exemples d'arguments* `sol` *et* `cle_l` *suivant les spécifications ci-dessous. Ces exemples doivent comporter une seule ligne* (`nl` $= 1$) *et au plus 5 colonnes* (`nc` ≤ 5).

1. *Donner et expliciter deux exemples pour lesquels la fonction renvoie* `False` *à la ligne 13 (pour deux raisons différentes).*
2. *Donner un exemple pour lequel le résultat de la fonction* `verif_ligne` *est incorrect. Expliquer comment la modifier pour obtenir un résultat correct.*

Partie II - Résolution systématique

On cherche maintenant à construire une solution du logimage. On considère une **solution partielle** `sol_p`, représentée par une liste de listes (comme précédemment, telle que `sol_p[i][j]` représente l'état de la case de la grille sur la ligne `i` et la colonne `j`), mais où toutes les cases ne sont pas encore déterminées à 0 ou à 1. Pour ceci, on codera par l'entier -1 la valeur initiale représentant une case non encore déterminée. Dans les schémas suivants, nous représenterons une telle case de couleur indéterminée par un point d'interrogation, comme illustré ci-dessous :

valeur -1 : ? valeur 0 : valeur 1 :

Une première méthode pour chercher une solution à partir des clés de lignes et de colonnes consiste à tester toutes les grilles possibles une par une. On suppose ici qu'on dispose d'une fonction `verif(sol_p,cle_l,cle_c)` qui renvoie `True` si et seulement si la solution `sol_p` est une solution complète et valide du logimage décrit par `cle_l` et `cle_c`, et dont la complexité est en $O(\texttt{nl} \times \texttt{nc})$. Nous allons remplir les cases ligne après ligne de haut en bas et de gauche à droite.

Question 5. *On suppose que les n premières cases de la grille sont déjà fixées à 0 ou à 1 ; ce sont les cases des k premières lignes (le plus en haut), ainsi que les ℓ cases les plus à gauche de la $(k+1)^{ème}$ ligne. Exprimer k et ℓ à partir de la valeur de n.*

Question 6. *Écrire une fonction* `liste_solutions` *qui prend pour argument les clés des lignes et des colonnes* `cle_l` *et* `cle_c` *et qui teste systématiquement toutes les combinaisons possibles de 0 et 1 des cases de la grille, en utilisant la fonction* `verif` *sur chacune d'entre elles, pour renvoyer la liste de toutes les solutions valides. On utilisera une fonction auxiliaire récursive* `liste_solutions_aux` *qui prend comme arguments un entier* `n`*, une solution partielle* `sol_p` *ainsi qu'une liste de solutions* `liste` *et qui ajoute à* `liste` *toutes les solutions valides dont les* `n` *premières cases (au sens de la question précédente) sont celles dont la valeur est fixée dans* `sol_p`*. Donner la complexité de la fonction* `liste_solutions`*.*

On peut diminuer le nombre de grilles testées en arrêtant l'exploration quand une ligne (ou une colonne) contient plus de cases noires que ce qui est dicté par la liste de ses clés.

Question 7. *Comment modifier la fonction de la question 6 pour ne pas explorer les grilles contenant trop de cases noires sur au moins une ligne ou une colonne ? On pourra se contenter d'indiquer les modifications à faire dans le code.*

Partie III - Placements possibles d'un bloc

Les algorithmes de la partie précédente ont une complexité trop élevée pour permettre de résoudre des grilles de grande taille. Dans cette partie, on considère un bloc particulier et on cherche à savoir où il peut être placé dans une ligne, en fonction des cases déjà déterminées dans cette ligne (cases blanches ou noires). Dans toute cette partie, on s'intéresse à une seule ligne de la grille, d'indice `i_ligne` fixé. Cet indice et la solution partielle `sol_p` pourront être considérés comme des variables globales accessibles à toutes les fonctions.

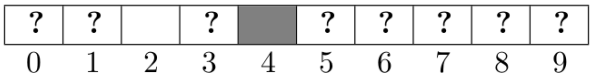
On cherche à placer un nouveau bloc de taille `s` dans la ligne d'indice `i_ligne`. On dit qu'un bloc de `s` cases **commence à l'indice** `j` s'il occupe les cases d'indice `j,j+1,...,j+s-1` de la ligne. Soit `c` l'indice d'une case dans la ligne considérée ($0 \leq$ `c` $<$ `nc`). On cherche d'abord à savoir si le bloc de taille `s` que l'on souhaite placer peut commencer à l'indice `c`, en fonction des décisions déjà prises, c'est-à-dire des valeurs de la liste `sol_p[i_ligne]` qui sont déjà fixées à 0 ou à 1. On dit qu'**une case** `i` de la ligne **est en conflit avec le placement du bloc en position** `c` si la valeur de `sol_p[i_ligne][i]` empêche de faire commencer le bloc en position `c`, c'est-à-dire si :

— (a) `i` $=$ `c` -1 et `sol_p[i_ligne][i]=1` (`i` est une case noire juste avant le bloc),

— (b) `c` $\leq$ `i` $<$ `c` $+$ `s` et `sol_p[i_ligne][i]=0` (`i` est une case blanche au milieu du bloc),

— (c) ou `i` $=$ `c` $+$ `s` et `sol_p[i_ligne][i]=1` (`i` est une case noire juste après le bloc).

Considérons par exemple la ligne de 10 cases représentée ci-dessous, avec deux cases prédéterminées (la case 4 est noire et la case 2 est blanche), dans laquelle on veut placer un bloc de taille `s` $= 3$. La case `i` $= 4$ est en conflit avec le placement du bloc commençant en position `c` $= 5$ à cause de (a). Pour un placement commençant en position `c` $= 1$, les cases `i` $= 2$ et `i` $= 4$ sont en conflit, respectivement à cause de (b) et (c). Pour un placement commençant en position `c` $= 3$, aucune case n'est en conflit.



Question 8. *Écrire une fonction* `conflit(c,s)` *qui, pour un bloc de taille* `s` *que l'on souhaite faire commencer à la case d'indice* `c`, *renvoie la plus petite position en conflit avec ce placement. S'il n'y a aucun conflit pour ce placement, la fonction doit renvoyer* `nc`. *On supposera que* `c` $+$ `s` $\leq$ `nc`. *La complexité doit être en* O(`s`) *et sera justifiée.*

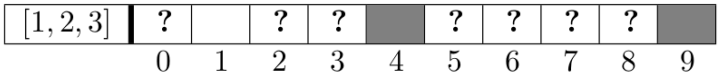
On veut maintenant trouver la **première position possible** d'un bloc, c'est-à-dire l'indice le plus petit auquel un bloc peut commencer. On cherche ici la première position possible d'un bloc **à partir d'un indice** `c` (donc supérieure ou égale à `c`). Dans l'exemple précédent, la première position possible pour placer un bloc de taille `s` $= 3$ à partir de l'indice `c` $= 1$ est 3. On pourrait utiliser la fonction conflit sur tous les indices à partir de `c`, avec une complexité O(`nc` $\times$ `s`), mais on veut une solution plus efficace.

Question 9. *Écrire une fonction* `prochain(c,s)` *qui renvoie la première position valide supérieure ou égale à* `c` *à laquelle peut commencer un bloc de taille* `s`, *et* -1 *si une telle position n'existe pas. La complexité de la fonction devra être en* O(`nc`) *et sera justifiée.*

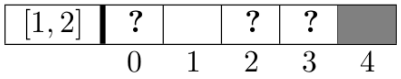
Partie IV - Placements possibles de tous les blocs d'une ligne

Comme dans la partie précédente, on se concentre ici sur le placement des blocs sur la ligne d'indice `i_ligne`. On pourra considérer que `i_ligne`, `sol_p` et `cle_l` sont des variables globales accessibles à toutes les fonctions.

On cherche maintenant à prendre en compte toutes les cases déjà coloriées en noir : si comme dans l'exemple ci-dessous, une case noire a été fixée en toute fin de ligne, alors cette case doit nécessairement être incluse dans le dernier bloc. Dans l'exemple ci-dessous, le dernier bloc (de taille 3) peut commencer uniquement à l'indice 7 afin qu'il contienne la case noire en position 9.



Nous allons procéder par programmation dynamique, en construisant une matrice `M` de taille `nc` $\times$ `B`, où `B` = `len(cle_1[i_ligne])` est le nombre de blocs sur la ligne considérée. Pour calculer la valeur de `M[c][b]`, on considère le sous-problème où il faut placer les blocs d'indice 0 à `b` dans la portion de ligne contenant les cases d'indice 0 à `c` de la ligne, en prenant en compte toutes les cases noires parmi elles. Pour l'exemple ci-dessus, `M[4][1]` correspond ainsi au sous-problème qui considère uniquement les cases 0 à 4 de la ligne, et cherche à y placer les blocs d'indice 0 et 1. Ce sous-problème est représenté ci-dessous :



Dans ce sous-problème, `M[c][b]` indique la **première position possible** pour le bloc d'indice `b`, **en prenant en compte toutes les cases noires** dans la portion de ligne considérée. Dans l'exemple ci-dessus, le bloc d'indice 1 (de taille 2) devra nécessairement inclure la case d'indice 4 car celle-ci est déjà coloriée en noir. On aura donc `M[4][1]` = 3. Lorsque les blocs d'indice 0 à `b` ne peuvent pas être placés dans les cases d'indice 0 à `c`, on pose par convention `M[c][b]` = -1.

On note `s` la taille du bloc d'indice `b` dont on veut calculer la première position possible : `s` = `cle_l[i_ligne][b]`. On considère tout d'abord le cas d'un bloc qui n'est pas le premier de la ligne (`b` > 0). On peut exprimer la valeur de `M[c][b]` de façon récursive, en considérant les deux cas suivants :

1. Si le dernier bloc pouvait être placé sans la dernière case (`M[c-1][b]` ≥ 0) et que la dernière case n'est pas noire (`sol_p[i_ligne][c]` $\neq 1$) alors on peut placer ce bloc de la même façon que sans la dernière case, c'est-à-dire `M[c][b]` = `M[c-1][b]`.
2. Sinon, on regarde si on peut placer le bloc en toute fin de ligne (donc le faire commencer à la case d'indice `c` $-$ `s` $+ 1$). Pour ceci, il faut que les deux conditions ci-dessous soient réunies :
   (a) `conflit(c-s+1,s)` $>$ `c` (il n'y a pas de conflit avec les cases de la ligne jusqu'à `c`)
   (b) `M[c-s-1][b-1]` ≥ 0 (il y a suffisamment de place dans les cases d'indice 0 à `c` $-$ `s` $- 1$ pour placer les blocs précédents, afin de laisser une case blanche à l'indice `c` $-$ `s`)

   Si ces deux conditions sont vérifiées, alors on peut placer le bloc en position `c` $-$ `s` $+1$, donc `M[c][b]` = `c` $-$ `s` $+1$. Sinon, le dernier bloc ne peut pas être placé : `M[c][b]` = -1.

Question 10. *On suppose que les valeurs de* `M[c][0]` *ont été précalculées pour tout* `c`. *Écrire un algorithme* `calcul_matrice` *qui prend* `M` *en argument et qui calcule le reste de la matrice par programmation dynamique. La complexité de la fonction devra être en* $\mathrm{O}\left(\mathtt{nc}^2\right)$ *et sera justifiée.*

Dans le cas du premier bloc (`b` = 0), une attention particulière doit être portée à la première case noire de la ligne, si elle existe. On note `p` la position de la première case noire si elle existe (`sol_p[i_ligne][p]` = 1 et `sol_p[i_ligne][j]` $\neq 1$ pour tout `j` $<$ `p`) ou `p` = -1 s'il n'y a pas de case noire dans `sol_p`.

Question 11. *Expliquer en quoi la position de la première case noire contraint la valeur de* `M[c][0]`. *En prenant en compte cette contrainte, modifier le calcul de* `M[c][b]` *détaillé ci-dessus pour le cas* `b` $= 0$. *On exprimera* `M[c][0]` *en fonction de* `sol_p[i_ligne]`, *des valeurs de* `M[c'][0]` *pour* `c'` $<$ `c`, *de* `p` *et du résultat de* `conflit(c-s+1,s)`.

Question 12. *Écrire une fonction* `premiere_case` *qui prend en argument la matrice* `M` *et qui renvoie une liste de la même taille que* `cle_l[i_ligne]` *dont l'élément d'indice* `b` *est la première position possible du bloc décrit par* `cle_l[i_ligne][b]`, *étant donnés toutes les cases prédéterminées et tous les autres blocs de la ligne. S'il n'existe pas de solution, alors la fonction renverra une liste vide. On pourra déterminer les premières positions possibles des blocs en commençant par ceux les plus à droite (donc d'indices les plus grands).*

On appelle `liste_pp` la **liste des premières positions** calculée dans la question précédente. De façon similaire, on peut calculer `liste_dp`, la **liste des dernières positions** possibles pour le **début de chaque bloc**, c'est-à-dire le plus grand indice `c` auquel peut **commencer** chaque bloc. Lorsque la première et la dernière position d'un bloc sont proches, on peut déterminer avec certitude que certaines cases appartiennent à ce bloc et doivent donc être coloriées en noir. Par exemple, pour la ligne représentée ci-dessous avec ses clés, nous obtenons `liste_pp=[0,5,10]` et `liste_dp=[5,8,13]`. Le second bloc commence donc au plus tôt à la case 5, et au plus tard à la case 8. Comme ce bloc est de taille 4, il contiendra nécessairement la case 8, que l'on peut donc colorier en noir.

$[2,4,1]$	?	?		?		?	?	?	?	?	?	?	?	?
	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Question 13. *Écrire une fonction* `remplissage` *qui prend en arguments les listes* `liste_pp` *et* `liste_dp` *et colorie en noir toutes les cases de la ligne* `sol_p[i_ligne]` *qui peuvent être déduites de ces deux listes par le critère précédent. On suppose que les listes* `liste_pp` *et* `liste_dp` *ne sont pas vides, c'est-à-dire qu'il existe une solution et que la ligne contient au moins un bloc.*

On peut déduire d'autres propriétés des listes `liste_pp` et `liste_dp`. Si on reprend l'exemple précédent, la case 3 est entourée de deux cases à 0 (blanches), donc ne pourrait participer qu'à un bloc de taille 1. Les deux listes `liste_pp` et `liste_dp` nous disent cependant que le seul bloc dont l'intervalle de positions contient cette case est le bloc 0 qui est de taille 2. On peut donc en déduire que la case 3 est blanche dans la solution. Plus généralement, pour une case d'indice i donné, on peut calculer la taille maximale `m` d'un bloc qui contiendrait cette case à partir des cases blanches les plus proches qui encadrent `i`. À l'aide des listes `liste_pp` et `liste_dp`, on sait quels blocs sont susceptibles de couvrir la case d'indice `i`. Si la taille minimale de ces blocs est plus grande que `m`, alors aucun bloc ne pourra couvrir la case d'indice `i`, qui est donc nécessairement blanche.

Question 14. *Écrire une fonction* `cases_blanches` *qui prend en arguments les listes* `liste_pp` *et* `liste_dp` *et colorie en blanc les cases de* `sol_p[i_ligne]` *qui ne peuvent faire partie d'aucun bloc d'après le critère précédent, avec les mêmes hypothèses sur les listes* `liste_pp` *et* `liste_dp` *que dans la question précédente.*

Fin du sujet.