Épreuve disciplinaire

Le sujet est constitué de 2 problèmes qui peuvent être traités de manière indépendante.

Notes de programmation : Vous disposez, pour répondre aux questions de ce sujet, des fonctions Python de manipulation de listes ou de matrices suivantes :

- On peut créer une liste de taille n remplie avec la valeur x avec `li = [x] * n`.
- On peut obtenir la taille d'une liste `li` avec `len(li)`.
- Si `li` est une liste de n éléments, on peut accéder au k-ème élément (pour $0 \leq k < $ `len(li)`) avec `li[k]`. On peut définir sa valeur avec `li[k] = x`.
- On peut ajouter un élément `x` dans une liste `li` à l'aide de `li.append(x)`, et on considèrera qu'il s'agit d'une opération élémentaire.
- Les matrices sont des listes de listes, chaque sous-liste étant considérée comme une ligne de la matrice. Si `mat` est une matrice, elle possède `len(mat)` lignes et `len(mat[0])` colonnes.
- On peut créer une matrice de n lignes et p colonnes, dont toutes les cases contiennent x avec `mat = [[x for j in range(p)] for i in range(n)]`.
- On accède (resp. modifie) l'élément de `mat` dans la i-ème ligne et j-ème colonne avec `mat[i][j]` (resp. `mat[i][j] = x`).
- On peut concaténer deux listes en utilisant l'opération `li1 + li2`. On utilisera aussi cette opération dans des expressions mathématiques.
- `li[a:b]` désigne la liste des éléments d'indice compris entre a et $b-1$ dans `li`. On utilisera aussi cette opération dans des expressions mathématiques.

Les autres fonctions sur les listes (`sort`, `index`, `max`, etc.) sont interdites à moins de les réécrire explicitement. L'opérateur `in` d'appartenance à une liste est interdit, mais on peut utiliser ce mot-clé dans les autres contextes (par exemple dans une boucle `for`).

Complexité : Par *complexité* d'un algorithme, on entend le nombre d'opérations élémentaires nécessaires à l'exécution de cet algorithme dans le pire cas. Lorsque cette complexité dépend d'un ou plusieurs paramètres $k_0, \ldots, k_{r-1}$, on dit que la complexité est $O(f(k_0, \ldots, k_{r-1}))$ s'il existe une constante $C > 0$ telle que, pour toutes les valeurs $k_0, \ldots, k_{r-1}$ suffisamment grandes, ce nombre d'opérations élémentaires est majoré par $C \times f(k_0, \ldots, k_{r-1})$.

Problème 1 Programmation en Python

Ce problème porte sur l'écriture de programmes à l'aide du langage Python.

Partie I Programmes divers

▶ **Question 1** Écrire une fonction `fibonacci(n)` qui prend en argument un entier `n` supérieur ou égal à 2 et renvoie la liste des `n` premiers termes de la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$ définie par $F_0 = 0$ et $F_1 = 1$ et $\forall n \geq 2, F_n = F_{n-1} + F_{n-2}$ (chaque terme est la somme des deux précédents).

▶ **Question 2** Écrire une fonction `indice_min(li)` qui prend en argument une liste d'entiers `li` et renvoie l'indice d'un de ses minimums.

▶ **Question 3** Que renverra `indice_min([1, 0, 2, 0])` avec votre programme ?

▶ **Question 4** Écrire une fonction `zero_sur_lignes(mat)` qui prend en argument une matrice (c'est-à-dire une liste de listes) contenant des 0 et des 1, renvoie `True` lorsque chaque ligne contient au moins un zéro, et `False` sinon.

Note : un programme efficace sera valorisé.

▶ **Question 5** Écrire une fonction `lettre_majoritaire(ch)` qui prend en argument une chaîne de caractères non vide et renvoie le caractère qui apparait le plus fréquemment. Ainsi, `lettre_majoritaire('abcdedde')` devrait renvoyer `'d'`.

Note : l'utilisation efficace d'un dictionnaire sera valorisée. On pourra alors utiliser l'opérateur `in`.

Partie II Saut de valeur maximale

A. Introduction

Dans une liste de flottants `li`, on appelle *saut* un couple (i, j) avec $0 \leq i \leq j < \texttt{len(li)}$, et la *valeur* d'un saut est la valeur `li[j] - li[i]`. On va ici programmer plusieurs manières de trouver un saut de valeur maximale dans une liste. Par exemple, dans la liste `[2.0, 0.2, 3.0, 5.3, 2.0]`, un tel saut est `(1, 3)` (car 0.2 et 5.3 sont aux indices 1 et 3 respectivement).

▶ **Question 6** Écrire une fonction `valeur(li, saut)` qui prend en argument une liste et un saut et renvoie la valeur du saut.

▶ **Question 7** Donner un exemple de liste avec exactement deux sauts de valeur maximale et préciser ces sauts.

▶ **Question 8** À l'aide d'un contre-exemple, montrer qu'on ne peut pas se contenter de chercher le minimum et le maximum d'une liste pour trouver un saut de valeur maximale.

▶ **Question 9** Écrire une fonction `saut_max_naif(li)` qui renvoie un saut de valeur maximale en testant tous les couples (i, j) tels que $0 \leq i \leq j < \texttt{len(li)}$.

B. Programmation dynamique

On décrit ici un algorithme utilisant le paradigme de la programmation dynamique pour résoudre ce problème : pour chaque k entre 1 et `len(li)`, on va calculer m_k l'indice du minimum de `li[0:k]`, et le couple (i_k, j_k) un saut de valeur maximale dans `li[0:k]`. Ainsi, on aura $m_1 = i_1 = j_1 = 0$ car `li[0:1]` ne comporte qu'un seul élément.

▶ **Question 10** Pour $k <$ `len(li)`, expliquer comment on peut calculer efficacement m_{k+1} à partir de m_k et des valeurs dans `li`.

▶ **Question 11** Justifier que la relation suivante est correcte.

$$(i_{k+1}, j_{k+1}) = \begin{cases} (i_k, j_k) \text{ si } \texttt{li}[k] - \texttt{li}[m_k] < \texttt{li}[j_k] - \texttt{li}[i_k] \\ (m_k, k) \text{ sinon} \end{cases}$$

▶ **Question 12** Écrire une fonction `saut_max_dynamique(li)` qui renvoie un saut de valeur maximale en utilisant la relation de la question 11.

▶ **Question 13** Déterminer la complexité de votre programme dans le pire cas, puis comparer cette complexité avec celle du programme donné en question 9.

C. Méthode diviser pour régner

On propose une dernière méthode pour le calcul d'un saut de valeur maximale utilisant le paradigme diviser pour régner. On notera $\lfloor x \rfloor$ la partie entière d'un nombre x quelconque.

Si une liste `li` comporte $n \geq 2$ éléments, on souhaite calculer :

- (i_g, j_g) un saut de valeur maximale lorsque $j_g < \lfloor n/2 \rfloor$,
- (i_d, j_d) un saut de valeur maximale lorsque $i_d \geq \lfloor n/2 \rfloor$,
- et (i_m, j_m) un saut de valeur maximale lorsque $i_m < \lfloor n/2 \rfloor \leq j_m$.

▶ **Question 14** Justifier qu'un saut de valeur maximale de `li` est nécessairement un des trois ci-dessus.

▶ **Question 15** Avec les notations précédentes, justifier que i_m est nécessairement l'indice d'une valeur minimale dans la moitié gauche de `li` (on admettra que, de même, j_m est nécessairement l'indice d'une valeur maximale dans la moitié droite de `li`).

▶ **Question 16** Écrire une fonction récursive `saut_max_aux(li, a, b)` qui prend en argument une liste `li` et deux indices $a < b$ de cette liste et renvoie le quadruplet constitué d'un saut de valeur maximale dans `li` entre les indices a et $b-1$ (deux valeurs), des indices d'un minimum et d'un maximum de `li` entre les indices a et $b-1$ (deux valeurs).

Ainsi, `saut_max_aux([2.0, 5.0, 3.0, 4.0, 6.0, 1.0], 2, 6)` doit renvoyer `(2, 4, 5, 4)`, car entre les indices 2 (inclus) et 6 (exclus), le saut de valeur maximale est $(2, 4)$ (de valeur 3.0), le minimum est 1.0 et le maximum est 6.0.

Note : l'efficacité de l'algorithme proposé sera valorisée.

▶ **Question 17** En déduire une fonction `saut_max(li)` qui renvoie un saut de valeur maximale d'une liste `li`.

Problème 2 Résolution de logigrammes

Partie III Cases, blocs et indications

A. Indications

On considère une ligne `li` constituée de cases qui peuvent être blanches ou noires. On notera $|\texttt{li}|$ la taille d'une telle ligne.

Les lignes sont représentées en Python comme des listes contenant des 0 (pour indiquer une case blanche) ou des 1 (pour indiquer une case noire).

Un *bloc* dans cette ligne est une succession maximale de cases noires, c'est-à-dire qu'un bloc est toujours précédé et suivi par une case blanche, ou une extrémité de la ligne. La *longueur* d'un bloc est le nombre de cases noires qu'il contient. La *valeur* d'une ligne `li` est le nombre total de cases noires qu'elle contient, que l'on notera $v(\texttt{li})$. L'*indication* de cette ligne est la liste ordonnée de gauche à droite des longueurs de tous les blocs de la ligne, comme illustré sur la figure 1. On notera cette indication $\chi(\texttt{li})$, et on appellera *valeur d'une indication* (notée également $v(\chi(\texttt{li}))$) la somme de ses éléments.

Dans toute cette partie, on notera $|\chi(\texttt{li})|$ le nombre d'éléments dans son indication.

```
ligne_ex = [1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0]
indication_ex = [3, 5, 1]
```

Figure 1: exemple d'une ligne et son indication. La ligne est de *taille* 15 et de *valeur* 9.

▶ **Question 18** Donner un exemple d'une ligne distincte de `ligne_ex` (donné en figure 1) mais ayant la même taille et la même indication.

▶ **Question 19** Écrire une fonction `valeur_ligne(li)` qui prend en argument une ligne `li` et renvoie sa valeur $v(\texttt{li})$ comme définie ci-dessus.

▶ **Question 20** Écrire une fonction `indication(li)` qui prend en argument une ligne `li` et renvoie son indication $\chi(\texttt{li})$. La fonction proposée devra avoir une complexité en $O(|\texttt{li}|)$.

▶ **Question 21** Justifier que pour toute ligne `li`, on a $v(\texttt{li}) \leq |\texttt{li}| + 1 - |\chi(\texttt{li})|$, et donner un exemple où il y a égalité.

B. Lignes réduites

Dans cette sous-partie, on fixe une longueur n et une indication $\mathcal{I}$, et on souhaite trouver toutes les lignes de taille n ayant cette indication.

Étant donnée une ligne `li` telle que $\chi(\texttt{li}) = \mathcal{I}$, on construit la *ligne réduite* $\phi(\texttt{li})$ de la manière suivante :

- on réduit la taille de chaque bloc de `li` à 1 en supprimant autant de cases noires qu'il le faut,
- puis, dans chaque intervalle de cases blanches entre deux blocs, on supprime exactement une case blanche. On ne modifie pas les intervalles de cases blanches aux extrémités.

▶ **Question 22** Donner la ligne réduite $\phi(\texttt{ligne\_ex})$ de la ligne de la figure 1.

▶ **Question 23** Écrire une fonction `ligne_reduite(li)` qui prend en argument une ligne `li` et renvoie la ligne réduite $\phi(\texttt{li})$.

▶ **Question 24** Pour une ligne `li` quelconque, exprimer la taille de la ligne réduite $|\phi(\texttt{li})|$ en fonction de $|\texttt{li}|$ et de $v(\texttt{li})$. Vous justifierez votre réponse.

▶ **Question 25** Soient k un entier, $\mathcal{I}$ une indication, et $\texttt{li}_r$ une ligne de taille k possédant exactement $|\mathcal{I}|$ cases noires. Montrer que $\texttt{li}_r$ est la ligne réduite d'une ligne de taille $k + v(\mathcal{I}) - 1$ et d'indication $\mathcal{I}$.

▶ **Question 26** Écrire une fonction `reconstruire_ligne(li_r, indic)` qui prend en argument une ligne réduite `li_r` et une indication `indic`, et renvoie une ligne dont la ligne réduite est `li_r` et dont l'indication est `indic`, lorsqu'une telle ligne existe. Si aucune ligne ne convient, aucun comportement particulier n'est exigé.

▶ **Question 27** Soient deux lignes l_1 et l_2 de même taille ($|l_1| = |l_2|$), même indication ($\chi(l_1) = \chi(l_2)$), et même ligne réduite ($\phi(l_1) = \phi(l_2)$). Montrer que $l_1 = l_2$.

▶ **Question 28** Soient n un entier et $\mathcal{I}$ une indication. On note E l'ensemble des lignes de taille n et d'indication $\mathcal{I}$. Déduire des questions précédentes que la fonction ϕ réalise une bijection entre E et un autre ensemble que l'on précisera, et en conclure que E est de cardinal $\binom{n - v(\mathcal{I}) + 1}{|\mathcal{I}|}$, où $\binom{n}{p}$ désigne le nombre de combinaisons de p éléments parmi n (à savoir $\frac{n!}{p!(n-p)!}$).

C. Énumeration de lignes

Soient $\mathcal{I}$ une indication et k un entier. On souhaite énumérer toutes les lignes réduites de taille k compatibles avec $\mathcal{I}$, c'est-à-dire toutes les listes de taille k, contenant exactement $|\mathcal{I}|$ cases noires et $k - |\mathcal{I}|$ cases blanches. On admettra qu'il est possible d'énumérer toutes ces lignes réduites en commençant par celle dont les 0 sont à gauche (c'est-à-dire qu'elle est de la forme `[0, ..., 0, 1, ..., 1]`) puis en passant d'une ligne réduite à la suivante à l'aide de la fonction `next(li)` donnée ci-dessous.

Note : on ne demande pas de prouver que cette fonction est correcte.

```
def next(li):
    """Renvoie la ligne réduite suivant li dans l'ordre
    d'énumération des lignes réduites."""
    k = len(li)
    for i in range(k-1, 0, -1):
        if li[i] == 1 and li[i-1] == 0:
            return li[0:i-1] + [1, 0] + li[k-1:i:-1]
    return []
```

▶ **Question 29** Que renvoie `next([0, 0, 1, 1, 1])` ? Que renvoie `next([1, 1, 1, 0, 0])` ?

▶ **Question 30** Compléter la fonction `liste_lignes_reduites(nb_noires, k)`, donnée ci-dessous, qui renvoie la liste des lignes réduites de taille k ayant `nb_noires` cases noires.

Note : vous écrirez les lignes 6, 7 et 8 de la fonction sur votre copie.

```
1. def liste_lignes_reduites(nb_noires, k):
2.     # Première ligne réduite
3.     li = [0] * (k-nb_noires) + [1] * nb_noires
4.     # Liste des lignes réduites
5.     res = []
6.     while ...:
7.         res.append(...)
8.         li = ...
9.     return res
```

▶ **Question 31** Écrire une fonction `liste_lignes(indic, n)` qui renvoie la liste des lignes de longueur n et d'indication `indic`.

Partie IV Jeu du logigramme

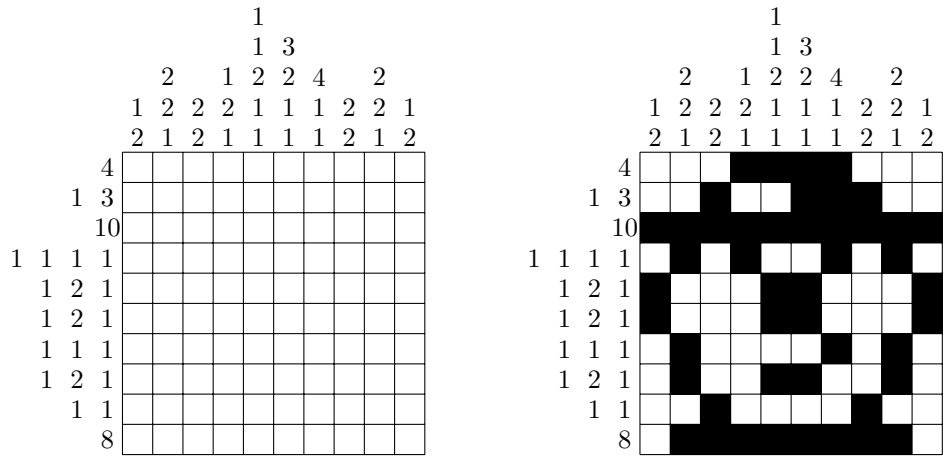


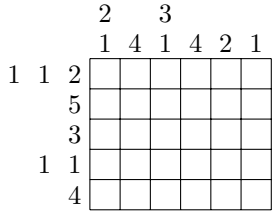
Figure 2: Exemple de *puzzle* et sa résolution pour le jeu du logigramme.

Un *puzzle* est la donnée de deux listes dont les éléments sont des indications (on rappelle qu'une indication est une liste d'entiers ; autrement dit un puzzle est un couple de listes de listes d'entiers).

Si on note n et p le nombre d'indications dans chaque liste respectivement, le jeu consiste à remplir une matrice de n lignes et p colonnes avec des cases noires ou blanches, de telle sorte que chaque ligne et chaque colonne correspondent à l'indication associée (voir figure 2).

En Python, les indications sur les lignes `indic_lignes` seront données sous forme d'une liste des indications sur les lignes, lues de haut en bas. De même pour `indic_colonnes` pour les colonnes,

lues de gauche à droite. Le puzzle est donné sous forme du tuple (`indic_lignes, indic_colonnes`) comme sur l'exemple de la figure 3.




```
puzzle_ex = (
    [[1, 1, 2], [5], [3], [1, 1], [4]],
    [[2, 1], [4], [3, 1], [4], [2], [1]]
)
```

Figure 3: Exemple du puzzle `puzzle_ex` et sa représentation en Python.

▶ **Question 32** Donner un exemple de puzzle avec $n = p = 2$ qui n'admette aucune solution.

▶ **Question 33** Donner un exemple de puzzle avec $n = p = 2$ qui admette plusieurs solutions, et donner toutes ses solutions.

Une grille de solution sera représentée par une matrice à n lignes et p colonnes, et on inscrit dans chaque case :

- un 0 si la case est blanche ;
- un 1 si la case est noire ;
- et -1 enfin si la couleur de la case n'est pas encore déterminée.

▶ **Question 34** Écrire une fonction `grille_vide(puzzle)` qui prend en argument un puzzle et renvoie une grille ayant la bonne taille, mais ne contenant que des -1 (toutes les cases sont indéterminées).

▶ **Question 35** Écrire une fonction `nb_cases_ind(grille)` qui prend en argument une grille et renvoie le nombre de cases indéterminées dans la grille.

▶ **Question 36** Écrire une fonction `est_valide(puzzle, grille)` qui prend en entrée un puzzle et une grille de dimension adaptée dont toutes les cases sont déterminées, et renvoie `True` si la grille est une solution du puzzle, et `False` sinon.

▶ **Question 37** Donner la complexité de la fonction `est_valide` en fonction de n et p, les dimensions de la grille.

Partie V Remplissage

On suppose qu'on dispose d'une grille partiellement remplie, c'est-à-dire de la bonne dimension et pouvant contenir des cases blanches, des cases noires, et des cases indéterminées. L'objectif de cette partie sera de compléter au maximum la grille en utilisant la technique suivante : on regarde toutes les manières de compléter une ligne (respectivement une colonne), et si une case est toujours coloriée de la même manière, on complète la grille en fixant cette couleur dans cette case. On rappelle qu'on peut générer la liste des lignes d'une longueur donnée et pour une indication donnée avec la fonction `liste_lignes(indic, n)` de la question 31.

▶ **Question 38** Écrire une fonction `liste_lignes_compatibles(indic, ligne_partielle)` qui prend en arguments une indication, une ligne partiellement remplie d'une grille, et renvoie la liste de toutes les lignes ayant comme indication `indic` et compatibles avec les cases déjà remplies dans `ligne_partielle`.

Ainsi, on devrait obtenir les valeurs suivantes, comme sur la figure 4.

```
>>> liste_lignes_compatibles([2, 2], [-1]*6)
[[1, 1, 0, 1, 1, 0],
 [1, 1, 0, 0, 1, 1],
 [0, 1, 1, 0, 1, 1]]
>>> liste_lignes_compatibles([2, 2], [1, -1, 0, -1, -1, -1])
[[1, 1, 0, 1, 1, 0],
 [1, 1, 0, 0, 1, 1]]
```

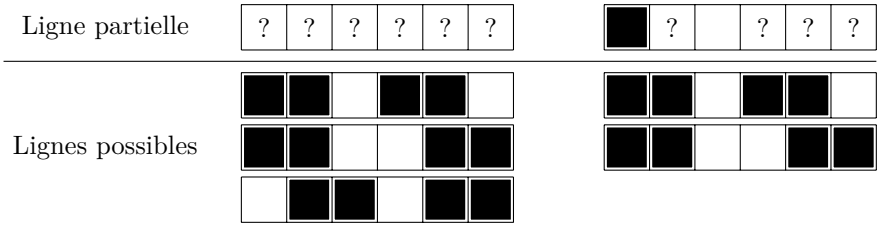


Figure 4: Liste des lignes compatibles avec l'indication `[2, 2]` et une ligne partielle. Les points d'interrogation sont les cases indéterminées.

▶ **Question 39** Écrire une fonction `remplir_ligne(indic, ligne)` qui prend en argument une indication, une ligne partiellement remplie, et renvoie une copie de `ligne` complétée au maximum, comme dans l'exemple ci-dessous :

```
>>> remplir_ligne([2, 2], [-1]*6)
[-1, 1, -1, -1, 1, -1]
>>> remplir_ligne([2, 2], [1, -1, 0, -1, -1, -1])
[1, 1, 0, -1, 1, -1]
```

▶ **Question 40** Écrire une fonction `completer_lignes(puzzle, grille)` qui applique cette méthode sur chaque ligne et modifie `grille` en fonction. Cette fonction ne renvoie rien.

▶ **Question 41** On exécute le programme suivant (`puzzle_ex` est donné dans la figure 3) :

```
grille = grille_vide(puzzle_ex)
completer_lignes(puzzle_ex, grille)
```

Donner alors la valeur de `grille`.

▶ **Question 42** On suppose qu'on dispose d'une fonction `completer_colonnes(puzzle, grille)` au fonctionnement similaire à la fonction `completer_lignes`. Écrire une fonction `completer(puzzle, grille)` qui applique ces deux fonctions sur la grille, jusqu'à ce que la grille ne soit plus modifiée.

Partie VI Retour sur trace

On s'intéresse dans cette partie à un algorithme plus général permettant de résoudre n'importe quelle grille.

▶ **Question 43** Que donnerait la fonction `completer` sur le puzzle `([[1, 1], [2]], [[1], [1], [1], [1]])` et avec une grille vide ? Donner sans justification l'unique solution de ce puzzle.

▶ **Question 44** La méthode du retour sur trace (ou *backtracking*) consiste, quand la méthode précédente (implémentée par la fonction `completer`) ne permet pas de résoudre une grille, à faire une hypothèse sur une case indéterminée de la grille : on essaye de terminer la résolution en supposant cette case blanche, puis en supposant cette case noire. Il est possible que cette hypothèse ne suffise pas, on formulera alors une autre hypothèse sur une autre case. Il est possible qu'une suite d'hypothèses aboutisse à une impasse, ce qui se traduira par la fonction `remplir_ligne` qui renvoie une liste vide. Dans ce cas, il convient d'abandonner cette suite d'hypothèses et d'en tester une autre.

Programmer la méthode du retour sur trace pour ce problème.

Note 1 : cette question est relativement ouverte, les réponses partielles seront valorisées si elles sont pertinentes.

Note 2 : l'algorithme décrit dans la partie V n'est pas le plus efficace : la génération des lignes possibles est ici exponentielle alors qu'on peut déterminer les cases nécessairement noires ou blanches avec une complexité cubique en la taille de la ligne. Aucune méthode n'est en revanche connue pour résoudre les puzzles en temps polynomial, ce problème fait partie des problèmes NP-complets.