DS 3 d'informatique de l'année 2024/2025 - MPSI - Polyominos

Consignes :

- ⋆ Les calculatrices sont interdites. Numérotez vos feuilles et faites apparaître les questions dans l'ordre du sujet.
- ⋆ Si vous repérez une erreur d'énoncé, signalez-le sur votre copie et poursuivez votre composition.
- ⋆ Les figures se trouvent sur la dernière page de ce document.

1. Introduction

Définition 1 (Informelle)**.** Soit $n \in \mathbb{N}^*$. Un ***n-omino*** est une forme géométrique connexe composée de n carrés unitaires. Le terme "connexe" signifie que tous les carrés composant la figure sont connectés.

Exemple 2. Il existe un seul 1-omino (figure 1), deux 2-ominos (figure 2) et six 3-ominos (figure 3). La figure 4 est un 5-omino, la figure 5 est un 12-omino, la figure 6 représente deux 7-ominos et la figure 7 est un 12-omino. De plus, la forme de la figure 8 n'est pas un 4-omino car elle n'est pas connexe.

Définition 3. Une forme géométrique est un ***polyomino*** s'il existe un entier $n \in \mathbb{N}^*$ tel que cette forme soit un n-omino.

Exemple 4. Les formes des figures 1 à 7 sont des polyominos.

Définition 5. En Python, un n-omino est une liste de listes de booléens « `P: list[list[bool]]` » vérifiant les conditions $(\mathcal{C}_1)$, $(\mathcal{C}_2)$, $(\mathcal{C}_3)$ et $(\mathcal{C}_4)$:

- $(\mathcal{C}_1)$ `P` est non vide et ses sous-listes sont toutes de la même taille.
- $(\mathcal{C}_2)$ Exactement n cases de `P` contiennent `True`, les autres cases contiennent `False`.
- $(\mathcal{C}_3)$ Au moins une des cases de la première ligne de `P` contient un `True`. Idem pour la première colonne, dernière ligne et dernière colonne.
- $(\mathcal{C}_4)$ L'ensemble des cases de `P` contenant `True` est connexe. Notez que le terme "connexe" sera défini formellement dans la partie 2.b..

Exemple 6. Les polyominos de la figure 6 sont représentés par :

```
P1_fig6 = [                          P2_fig6 = [
    [True , True , True ],               [False , True , True , True ],
    [True , False, True ],               [True  , True , True , False],
    [True , False, True ]]               [True  , False, False, False]]
```

1. Donner la représentation Python du 12-omino de la figure 7.

2. Vérification des propriétés $(\mathcal{C}_1)$ à $(\mathcal{C}_4)$

Le but de cette partie est de tester si une liste « `P: list[list[bool]]` » vérifie les conditions $(\mathcal{C}_1)$ à $(\mathcal{C}_4)$. Ainsi, sauf indication contraire, on ne suppose pas que `P` satisfait l'une de ces quatre conditions.

2. Écrire une fonction `afficher(P: list[list[bool]]) -> NoneType` qui affiche `P` en remplaçant les cases contenant `True` par le caractère `"#"` et les cases contenant `False` par le caractère `"-"`. Voici un exemple pour `P` ainsi que l'affichage correspondant :

```
P = [[True , False, False, True , True , True , True , False],
     [False, False, True , False, False, True ],
     [],
     [True , False, True , False, True ],
     [False, True , True , True ]]
```

```
#--####-
--#--#

#-#-#
-###
```

2.a. Conditions $(\mathcal{C}_1)$, $(\mathcal{C}_2)$ et $(\mathcal{C}_3)$

3. Écrire une fonction `test1(P: list[list[bool]]) -> bool` qui renvoie `True` si `P` vérifie $(\mathcal{C}_1)$ et `False` sinon.
4. (a) Écrire une fonction `cptTrue(P: list[list[bool]]) -> int` qui renvoie le nombre de cases de `P` contenant `True`.
   (b) En déduire une fonction `test2(n: int, P: list[list[bool]]) -> bool` qui renvoie `True` si `P` vérifie $(\mathcal{C}_2)$ et `False` sinon.
5. Dans cette question, on pourra supposer sans le vérifier que `P` satisfait $(\mathcal{C}_1)$.
   (a) Écrire une fonction :

   `ligne(P: list[list[bool]], i0: int) -> list[int,int]`

   qui renvoie la liste de tous les couples `(i,j)` tels que `i0 = i` et `P[i][j] = True`. On pourra supposer sans le vérifier que `i0` est un indice valide pour `P`.
   (b) ***À l'aide d'une fonction récursive***, écrire une fonction :

   `colonne(P: list[list[bool]], j0: int) -> list[int,int]`

   qui renvoie la liste de tous les couples `(i,j)` tels que `j0 = j` et `P[i][j] = True`. On pourra supposer sans le vérifier que `j0` est un indice valide pour `P`.
   (c) En déduire une fonction `test3(P: list[list[bool]]) -> bool` qui renvoie `True` si `P` vérifie $(\mathcal{C}_3)$ et `False` sinon.

2.b. Condition $(\mathcal{C}_4)$

On souhaite maintenant vérifier la condition $(\mathcal{C}_4)$. Dans un premier temps, il faut donner une définition formelle de la "connexité".

Notation 7. *Dans cette partie, la variable `P` est une liste de type « `list[list[bool]]` » vérifiant les conditions $(\mathcal{C}_1)$ à $(\mathcal{C}_3)$.*

Notation 8. *On note* $\mathtt{I} \subset \mathbb{N}^2$ *l'ensemble des couples* $\mathtt{(i,j)} \in \mathbb{N}^2$ *tels que* `P[i][j]` *existe et vaut* `True`.

Définition 9. Pour tout $\mathtt{(i,j)} \in \mathtt{I}$, un ***voisin*** de `(i,j)` est un élément de l'ensemble :

$$\Big\{ \mathtt{(i+1,j)}, \mathtt{(i-1,j)}, \mathtt{(i,j+1)}, \mathtt{(i,j-1)} \Big\} \cap \mathtt{I}$$

En d'autres termes, les 4 voisins possibles pour `(i,j)` sont les cases directement en dessous, au-dessus, à droite et à gauche de `(i,j)`.

Définition 10. Soit $\mathtt{k} \in \mathbb{N}$, $\mathtt{u} \in \mathtt{I}$ et $\mathtt{v} \in \mathtt{I}$. Un ***chemin de longueur*** `k` entre `u` et `v` est un `(k+1)`-uplet $\mathtt{(w_0, w_1, \dots w_k)} \in \mathtt{I}^{k+1}$ tel que :

$$\mathtt{w_0 = u}, \qquad \mathtt{w_k = v}, \qquad \forall \mathtt{i} \in [\![0 ; k-1]\!] : \mathtt{w_{i+1}} \text{ est un voisin de } \mathtt{w_i}.$$

Définition 11. `P` est dit ***connexe*** si pour tout $\mathtt{u} \in \mathtt{I}$ et tout $\mathtt{v} \in \mathtt{I}$, il existe un chemin entre `u` et `v`.

Définition 12. Soient $\mathtt{u} \in \mathtt{I}$ et $\mathtt{v} \in \mathtt{I}$ tels qu'il existe un chemin entre `u` et `v`. La ***distance*** entre `u` et `v` est le plus petit entier $\mathtt{k} \in \mathbb{N}$ tel qu'il existe un chemin de longueur `k` entre `u` et `v`.

Exemple 13. On s'intéresse aux deux 7-ominos de la figure 6. Pour le polyomino de gauche on pose $\mathtt{u_0 = (0,0)}$ et pour le polyomino de droite on pose $\mathtt{u_0 = (1,0)}$. Lorsqu'on calcule les distances entre $\mathtt{u_0}$ et tous les $\mathtt{v} \in \mathtt{I}$, on obtient les valeurs de la figure 9.

6. Recopier la figure 7 et y inscrire les distances entre $\mathtt{u_0 = (0,0)}$ et tous les $\mathtt{v} \in \mathtt{I}$.

Notre objectif est maintenant de vérifier si `P` est connexe. Pour cela, on fixe arbitrairement $u_0 \in I$, puis on explore de proche en proche toutes les cases accessibles. Plus précisément, nous allons utiliser deux variables :

→ Une variable « `vu: dict[(int, int): NoneType]` » contenant un dictionnaire dont les clés sont tous les couples `(i,j)` tels que `(i,j)` est une case ayant déjà été explorée. Les valeurs du dictionnaire seront toutes égales à `None` car nous ne les utiliserons pas.

→ Une variable « `dist: list[list[(int,int)]]` » telle que `dist[d]` est la liste de tous les $v \in I$ tels que u_0 et v sont à distance `d`.

L'algorithme que nous utiliserons s'appelle un ***parcours en largeur***. En voici le principe :

⋆ Initialement, on pose `dist = [[u]]`.

⋆ À chaque étape :
- On parcourt tous les couples `(i,j)` dans `dist[-1]` et on note `L` la liste de toutes les cases voisines des `(i,j)` n'ayant pas encore été explorées.
- On ajoute `L` dans `dist`.

⋆ La procédure s'arrête lorsque `dist[-1]` est vide. Pour tester si `P` est connexe, il suffit alors de vérifier si le nombre de cases explorées est égal au nombre d'éléments dans `I`.

Exemple 14. On s'intéresse aux deux 7-ominos de la figure 6. Pour le polyomino de gauche on pose u_0 `= (0, 0)` et pour le polyomino de droite on pose u_0 `= (1, 0)`. Voici la liste `dist` obtenue à la fin du parcours en largeur :

```
dist1_fig6 = [[(0, 0)],
              [(1, 0), (0, 1)],
              [(2, 0), (0, 2)],
              [(1, 2)],
              [(2, 2)],
              []]
```

```
dist2_fig6 = [[(1, 0)],
              [(2, 0), (1, 1)],
              [(0, 1), (1, 2)],
              [(0, 2)],
              [(0, 3)],
              []]
```

7. Donner la liste `dist` obtenue à la fin du parcours en largeur appliqué au 12-omino de la figure 7 avec u_0 `= (0, 0)`.
8. (a) Écrire une fonction `dansI(P: list[list[bool]], i: int, j: int) -> bool` qui indique si `(i, j)` $\in$ `I`. En particulier, si `i` et `j` ne sont pas des indices valides pour `P`, la fonction doit renvoyer `False`.

   (b) Écrire une fonction

   ```
   parcLarg(P: list[list[bool]], i0: int, j0: int) ->
   (dict[(int, int): NoneType], list[list[(int,int)]])
   ```

   qui applique le parcours en largeur sur `P` à partir de la case u_0 `= (i0, j0)`. Bien sûr, vous devez utiliser la procédure décrite ci-dessus. Les objets en sortie sont le dictionnaire « `vu` » et la liste « `dist` ». On pourra supposer sans le vérifier que `(i0, j0)` $\in$ `I`.

   (c) En déduire une fonction `test4(P: list[list[bool]]) -> bool` qui renvoie `True` si `P` vérifie $(\mathcal{C}_4)$ et `False` sinon.

3. Katamino

Le "Katamino" est un jeu de société où le joueur dispose d'un ensemble de polyominos et doit les utiliser pour paver un rectangle sans chevauchement. Par exemple la figure 10 montre comment paver un rectangle composé de 5 lignes et 9 colonnes en utilisant les 5-ominos de la figure 11. L'objectif de cette partie est d'écrire un programme Python pour résoudre ce problème.

Notation 15. *Dans cette partie, le type « `Polyo1` » désigne l'ensemble des listes « `P: list[list[bool]]` » vérifiant les conditions $(\mathcal{C}_1)$, $(\mathcal{C}_2)$, $(\mathcal{C}_3)$ et $(\mathcal{C}_4)$.*

9. À l'aide d'une compréhension de liste, écrire une fonction `convertir(P: Polyo1) -> list[int,int]` qui renvoie la liste de tous les couples `(i,j)` tels que `P[i][j] = True`. Votre fonction doit être composée d'exactement deux lignes : une avec le « `def` » et une avec le « `return` ».

Notation 16. *Dans la suite, on dira qu'une liste* `Q` *est de type « Polyo2 » s'il existe « P: Polyo1 » telle que* `Q = convertir(P)`.

3.a. Rotations et symétries

Lors d'une partie de Katamino, le joueur peut faire pivoter les pièces et les retourner.

Notation 17. *Soit P un polyomino. On note $T(P)$ l'ensemble des polyominos que peut obtenir le joueur en appliquant sur P des combinaisons :*

→ *De rotations de 90° dans le sens des aiguilles d'une montre.*

→ *De symétries par rapport à un axe vertical.*

10. Dessiner tous les éléments de $T(P)$ lorsque P est le polyomino de la figure 5.
11. Écrire une fonction `egal(P1: Polyo1, P2: Polyo1) -> bool` qui indique si `P1` et `P2` correspondent au même polyomino. En d'autres termes, votre fonction doit renvoyer le booléen « `P1 == P2` ». Les opérateurs de comparaison (`==`, `!=`, ...) peuvent être utilisés uniquement avec des objets de type `int` ou `bool` ; en particulier, écrire « `L1 == L2` » est interdit lorsque `L1` et `L2` sont des listes.
12. Écrire une fonction `sym(P: Polyo1) -> Polyo1` qui renvoie le polyomino obtenu en appliquant une symétrie sur `P` par rapport à un axe vertical.
13. Écrire une fonction `rot(P: Polyo1) -> Polyo1` qui renvoie le polyomino obtenu en appliquant une rotation de 90° dans le sens des aiguilles d'une montre sur `P`.
14. Déduire des questions précédentes une fonction : `makeT(P: Polyo1) -> list[Polyo1])` qui prend en entrée un polyomino P et renvoie $T(P)$. La liste obtenue ne doit pas contenir de doublon.

3.b. Recherche d'une solution du Katamino

Pour rechercher une solution au jeu du Katamino, on va adopter une approche récursive. L'idée est de partir de la grille vide notée G et d'y placer les polyominos les uns après les autres. Pour chaque polyomino P, on essaye successivement toutes les positions de P dans G. Lorsqu'une position valide est détectée, on y met P, puis on place récursivement les polyominos restants.

Notation 18. *On note « `P_list: list[Polyo1]` » la liste des polyominos que doit placer le joueur (aucune rotation/symétrie n'a encore été appliquée sur les éléments de* `P_list`*).*

Notation 19. *Dans la suite, le type « `Grille` » sera une abréviation pour « `list[list[int]]` ». De plus, pour tout objet* `G` *de type* `Grille` *:*

→ *Lorsque la case* `(i,j)` *est libre, on a* `G[i][j] = -1`.

→ *Lorsque le polyomino* `P_list[m]` *occupe la case* `(i,j)`*, on a* `G[i][j] = m`*. Notez qu'un polyomino* `P` *occupe* `cptTrue(P)` *cases.*

15. Soit `P = P_list[m]` un polyomino et `Q = convertir(P)`. Écrire une fonction

    `placer(G: Grille, Q: Polyo2, m: int, i: int, j: int) -> Grille` ou `NoneType`

    qui essaye de placer `P` dans `G` en faisant correspondre `P[0][0]` avec `G[i][j]`. Si `P` chevauche un autre polyomino, la fonction doit renvoyer `None` ; sinon elle doit renvoyer une **copie** de `G` dans laquelle les cases occupées par `P` sont remplacées par `m`.
16. Écrire une fonction récursive

    `katamino(m1: int, m2: int, P_list: list[Polyo1]) -> Grille`

    qui renvoie une solution au jeu du Katamino. L'entier `m1` (resp. `m2`) est le nombre de lignes (resp. colonnes) dans la grille.

4. Dénombrement des polyominos

Soit $n \in \mathbb{N}^*$ un entier fixé. Dans cette partie, on va voir deux méthodes différentes pour compter le nombre de n-ominos.

4.a. Méthode 1

Notation 20. *Pour tout $n \in \mathbb{N}^*$ et $m \in \mathbb{N}^*$, on note $E_{n,m}$ l'ensemble des listes « `L: list[int]` » telles que :*

→ `L` *est de taille m.*

→ *Tous les éléments de* `L` *appartiennent à $[\![0 ; n^2 - 1]\!]$.*

→ `L[0]` *appartient à $[\![0 ; n - 1]\!]$.*

→ `L` *est triée.*

Exemple 21. Avec $n = m = 2$:

$$E_{2,2} = \Big\{ \texttt{[0,1], [0,2], [0,3], [1,2], [1,3]} \Big\}$$

Exemple 22. Avec $n = 3$ et $m = 2$:

$$E_{3,2} = \Big\{ \begin{array}{l} \texttt{[0,1], [0,2], [0,3], [0,4], [0,5], [0,6], [0,7], [0,8],} \\ \texttt{[1,2], [1,3], [1,4], [1,5], [1,6], [1,7], [1,8],} \\ \texttt{[2,3], [2,4], [2,5], [2,6], [2,7], [2,8]} \Big\} \end{array}$$

17. Écrire une fonction `makeE(n: int, m: int) -> list[list[int]]` qui prend en entrée $n \in \mathbb{N}^*$ et $m \in \mathbb{N}^*$, et renvoie une liste contenant tous les éléments de $E_{n,m}$.
Indication : on pourra construire successivement les ensembles $E_{n,1}$, $E_{n,2}$, $E_{n,3}$, …, $E_{n,m}$.

Notation 23. *Soit $n \in \mathbb{N}^*$. On s'intéresse à une grille composée de n lignes et n colonnes dont les cases sont numérotées de 0 à $n^2 - 1$ (comme dans la figure 12 où $n = 5$). Pour tout $L \in E_{n,n}$, on note $\alpha(L)$ la grille où les cases numérotées par un élément de L sont marquées par une étoile.*

Exemple 24. La figure 13 représente $\alpha($`[1,7,9,16,23]`$)$

Exemple 25. La figure 14 représente $\alpha($`[2,10,12,15,22]`$)$

Exemple 26. La figure 15 représente $\alpha($`[0,5,6,7,10]`$)$

18. (a) Soit $n \in \mathbb{N}^*$. En utilisant les notations introduites dans cette partie, expliquer comment générer tous les n-ominos.
(b) En déduire une fonction `cptOminos(n: int) -> int` qui prend en entrée un entier $n \in \mathbb{N}^*$ et renvoie le nombre de n-ominos.

4.b. Méthode 2

La deuxième méthode pour générer les n-ominos s'appelle la ***méthode par croissance***. En voici le principe :

→ On part d'une grille contenant n lignes et n colonnes dans laquelle on va sélectionner des cases. Les cases sélectionnées formeront un ensemble connexe et représenteront donc un n-omino.

→ À chaque instant, chaque case c est dans l'état 0, 1 ou 2 :
- L'état 2 signifie que c a déjà été sélectionnée.
- L'état 1 signifie que c n'a pas encore été sélectionnée, mais qu'elle pourra l'être lors de la prochaine étape.
- L'état 0 signifie que c n'a pas été sélectionnée et qu'elle ne le sera pas lors de la prochaine étape.

→ Initialement, l'une des cases est dans l'état 1 et toutes les autres sont dans l'état 0. Ensuite, à chaque étape, on choisit une case c parmi celles dans l'état 1. La case c passe dans l'état 2 et tous ses voisins (au sens de la définition 9) qui sont dans l'état 0 passent dans l'état 1. Lorsque n cases sont sélectionnées, on obtient un n-omino.

→ Pour garantir que chaque n-omino n'est généré qu'une seule fois :
- La première case sélectionnée c_0 doit être sur la première ligne de la grille. Par la suite, aucune case de la première ligne située à gauche de c_0 ne peut être sélectionnée.

- On fait en sorte qu'au moins une case de la première colonne soit sélectionnée. Pour cela, on s'interdit de sélectionner une case si cela implique que la première colonne ne pourra pas être atteinte.
- Lorsqu'une case passe de l'état 0 à l'état 1, on lui attribue un numéro (égal au nombre de cases dans l'état 1 ou 2). À chaque étape, on s'interdit de sélectionner une case si son numéro est inférieur à celui de la case sélectionnée lors de l'étape précédente.

19. Écrire une fonction `cptOminosBis(n: int) -> int` qui prend en entrée un entier $n \in \mathbb{N}^*$ et renvoie le nombre de n-ominos. Bien sûr, vous devez utiliser la méthode par croissance décrite ci-dessus.

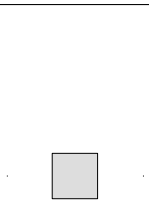

Figure 1

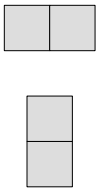

Figure 2

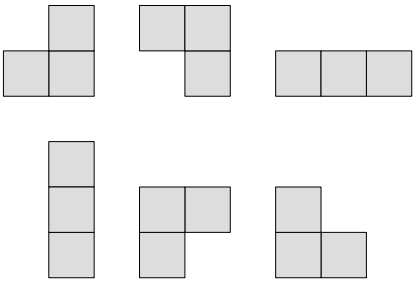

Figure 3

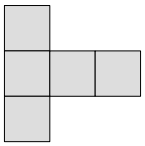

Figure 4

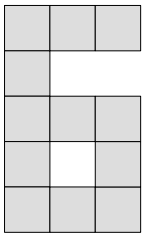

Figure 5

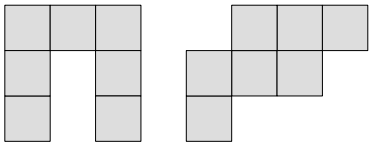

Figure 6

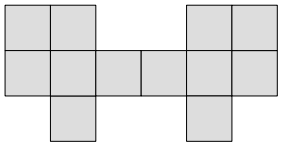

Figure 7

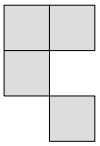

Figure 8

0	1	2
1		3
2		4

	2	3	4
0	1	2	
1			

Figure 9

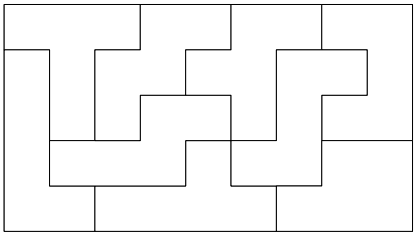

Figure 10

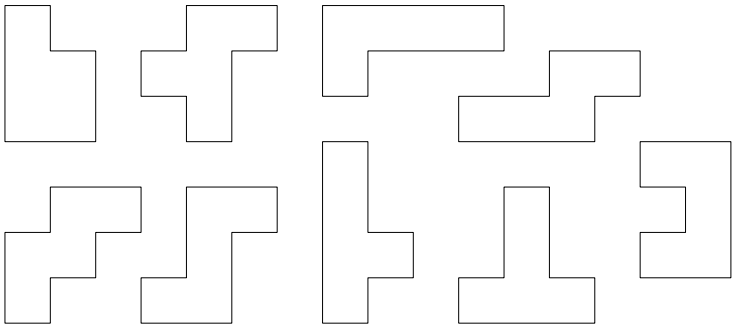

Figure 11

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Figure 12

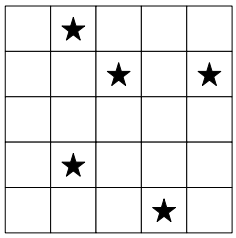

Figure 13

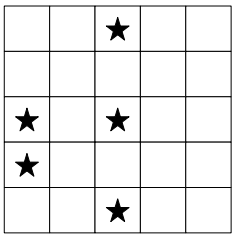

Figure 14

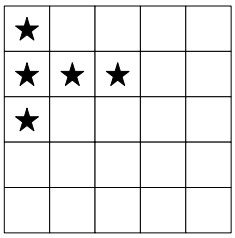

Figure 15