MINISTÈRE
DE L'ÉDUCATION
NATIONALE

Liberté
Égalité
Fraternité

CONCOURS GÉNÉRAL DES LYCÉES

—

SESSION 2026

—

Numérique et sciences informatiques

RAPPORT DE JURY

1 Composition du jury

— Marc de Falco, inspecteur général de l'éducation, du sport et de la recherche, président du jury.
— Pierre Hyvernat, maître de conférences, université Savoie Mont Blanc, vice-président du jury.
— Fanny Canivet, professeure, académie d'Amiens.
— Florian Hatat, professeur, académie de Versailles.
— Anne Héam, professeure, académie de Besançon.
— Mathias Hiron, président de France-ioi, organisateur des concours Algorea, Alkindi et Castor Informatique.
— Louis Jachiet, maître de conférences, Télécom Paris.
— Arnaud Lathelize, inspecteur d'académie, inspecteur pédagogique régional, académie de Nice.
— David Roche, professeur, académie de Grenoble.

2 Statistiques

Année	Candidats présents	Élèves suivant NSI en terminale
2022	576	16 158
2023	567	17 835
2024	531	17 612
2025	351	17 061
2026	392	15 055

Pour la session 2026, 516 candidates et candidats étaient inscrits, issus de 291 établissements, et 392 ont composé, soit 76 % des inscrits. Parmi les inscrits, on compte 48 candidates (9,3 %), alors que les filles représentent 15,5 % des élèves suivant l'enseignement de spécialité NSI en terminale à la rentrée 2025 (source : DEPP, note d'information n° 26-06). 356 inscrits (69 %) sont scolarisés dans un établissement public et 160 dans un établissement privé sous contrat ou homologué. Les académies les plus représentées sont celles de Versailles (59 inscrits), de Nantes (34), de Lyon (30), de Montpellier (29) et de Créteil (28). 40 inscrits sont scolarisés dans un établissement français à l'étranger.

3 Bilan de la session 2026

Le sujet comporte deux parties indépendantes. La première partie porte sur l'étude d'un jeu simple où un robot se déplace dans une grille selon certaines règles : cette partie permet de mobiliser des connaissances sur la programmation en Python, avec l'utilisation de programmation orientée objet ou de validation par jeux de tests, et des capacités de conception d'algorithmes. La deuxième partie étudie certains aspects usuels rencontrés dans l'écriture d'un compilateur : cette partie permet d'aborder des éléments de programmation avancée, avec plusieurs programmes assez longs à écrire et qui nécessitent de structurer le code, et des éléments d'analyse théorique des algorithmes.

Le sujet permet d'évaluer les candidates et les candidats aussi bien sur leur maitrise de la programmation en Python que sur leurs capacités de modélisation, de raisonnement et d'analyse abstraite.

Plusieurs parties sont indépendantes, ce qui permet aux candidates et aux candidats de continuer à progresser dans le sujet même en cas de blocage sur une partie. Chaque partie débutait par des questions assez simples pour se familiariser avec les concepts introduits par le sujet. Traiter ces questions est indispensable pour la compréhension du sujet, mais le jury apprécie surtout les copies présentant une progression nette dans les différentes parties. Il est peu valorisé de traiter uniquement les questions simples de début de partie.

4 Palmarès

Une grande partie des copies démontre un très bon niveau, voire un excellent niveau dans la discipline. Parmi les meilleures copies, le jury a retenu 18 copies qu'il a estimé être d'une qualité remarquable.

L'ensemble des copies témoigne d'un très bon niveau de préparation des candidates et des candidats.

Le jury félicite, cette année à nouveau, l'ensemble des candidates et des candidats pour leur participation et la bonne qualité générale des prestations. Il renouvelle sa reconnaissance aux professeurs pour la qualité de leur travail de préparation.

5 Conseils généraux aux futurs candidates et candidats

5.1 Présentation des copies

Le jury insiste particulièrement sur le soin qui doit être apporté à l'écriture et à la présentation. Le contenu de plusieurs copies a été très difficile à comprendre en raison d'une écriture peu lisible, de la présente de nombreuses ratures et de rajouts entre les lignes. Le jury conseille d'exploiter un brouillon pour aider à la réflexion et préparer les éléments de réponse avant leur rédaction.

À contrario, le jury note un effort de présentation sur plusieurs copies, effort qui va souvent de pair avec des raisonnements mieux construits et des programmes Python mieux structurés. La rédaction de programmes en Python nécessite souvent, dès que la réponse attendue est un peu longue, de réfléchir au préalable à la structure générale du programme et à bien prendre en compte tous les cas.

Le jury recommande aux candidats de bien faire attention à la numérotation des questions. Celle-ci doit respecter *strictement* la numérotation de l'énoncé. Toute modification introduite par les candidats (absence de numéro, confusion de questions, passage de la numérotation en chiffres romains) présente le risque de tromper la vigilance du jury et de conduire à une mauvaise interprétation de la réponse donnée.

Les parties indépendantes peuvent être traitées dans l'ordre que les candidats préfèrent. À l'intérieur d'une partie, les questions sont rarement indépendantes. Plusieurs candidats ont choisi de traiter dans le désordre les questions d'une même partie. Cette stratégie est rarement efficace car elle n'aide pas à comprendre le raisonnement attendu, et le résultat est souvent difficilement lisible.

5.2 Programmation

Le jury a apprécié le fait que la majorité des codes Python étaient correctement présentés. Il restera attentif sur ce point et invite les candidats des sessions suivantes à bien faire attention à la présentation, notamment à bien indenter les blocs de code. L'indentation des blocs de code doit être sans ambiguïté, tout en évitant les décalages vraiment trop grands qui conduisent à écrire des lignes de code compressées sur la marge droite de la copie.

Un code clairement écrit peut souvent se passer de commentaires, en particulier lorsque :

- les variables portent des noms explicites sur le rôle qu'elles jouent,
- le code n'est pas trop long. Ainsi, un code de fonction écrit sur plusieurs pages et sans aucun commentaire pour en indiquer la structure générale est très difficile à lire.

La clarté d'un code à la relecture fait partie des points évalués par le jury.

Le jury rappelle qu'il ne faut pas confondre les différents itérateurs disponibles sur les dictionnaires en Python : `for cle in dico:` ou son équivalent plus long à écrire `for cle in dico.keys():` ne doivent pas être confondus avec `for valeur in dico.values():`.

5.3 Notions de complexité attendues

Le formalisme théorique rigoureux de la complexité temporelle n'est pas un attendu de la spécialité, mais le programme attend que les candidats aient acquis une réflexion sur les questions de temps d'exécution.

On attend donc que les candidats aient une réflexion un peu avancée sur les points suivants :

- savoir distinguer des complexités constantes, linéaires, quadratiques ou plus grandes ;
- être sensibilisé aux complexités attendues pour les opérations élémentaires, notamment sur la différence entre `x in e` selon que `e` est une liste ou un dictionnaire ;
- programmer efficacement, en évitant des algorithmes évidemment inefficaces en temps.

6 Remarques détaillées par question

Question 1

Le sujet commence par la définition du problème, qu'il est important de comprendre pour aborder correctement la suite. Cette question de compréhension vise à assurer une entrée progressive dans le sujet.

Les questions 1b et 1c abordent sur des exemples la question de la terminaison des déplacements du robot, qui sera abordée dans un cas plus général en question 4.

Cette question a été traitée correctement dans la quasi-totalité des copies.

Question 2

Cette question permet d'évaluer à la fois quelques éléments simples de syntaxe Python en programmation orientée objet, ainsi que la compréhension des objets manipulés.

Il y a plusieurs manières d'obtenir une instanciation invalide : coordonnées positives mais en dehors de la grille, coordonnées négatives, numéro de portail présent 3 fois ou plus dans la liste, portail sur la première ou la dernière ligne, numéro de portail négatif ou supérieur à `(nb_lignes-2)*nb_colonnes-1`.

Question 3

Pour cette vraie première question de programmation du sujet, il faut bien faire attention à n'oublier aucune des conditions. De nombreuses copies oublient notamment de vérifier que les coordonnées sont valides ou qu'il n'y a bien qu'au plus deux portails de même numéro, ou bien qu'on ne place pas *deux portails dans une même case* de la grille.

Il est important de choisir une structure de données efficace pour vérifier que chaque numéro de portail apparait au plus deux fois. Un dictionnaire est adapté à cet usage, pour éviter d'autres tests linéaires en le nombre de portails ou en la taille de la grille.

Une manière standard de faire ce genre de validation peut être d'utiliser des *assertions* : `assert CONDITION` teste la condition et stoppe l'exécution si elle n'est pas vérifiée.

Toute autre méthode de validation, telle qu'une fonction auxiliaire qui renvoie un booléen, est acceptée tant qu'elle fait clairement la distinction entre les entrées valides et invalides.

Cette question distingue nettement les copies : certaines se contentent d'une vérification sommaire et incomplète, là où d'autres réussissent à vérifier efficacement toutes les conditions.

Question 4

Cette question est plutôt difficile : il faut déjà se convaincre, notamment en reprenant l'exemple de la question 1c et éventuellement en considérant d'autres exemples, que le robot ne peut pas rester bloqué sur la grille dans les hypothèses de la question.

La preuve formelle est ensuite difficile à rédiger. Le raisonnement est subtil, en particulier car, comme le montre la question 1c, un robot peut passer deux fois sur une même case contenant un portail : il peut rentrer une première fois sur un portail, et y ressortir à un autre moment.

Cette question a bien souvent été ignorée par les candidats, ou alors traitée de manière incorrecte. Le jury a accepté des réponses où l'argument n'est pas formellement complet ou entièrement rédigé mais où l'idée de terminaison est bien présente et où il y a un effort pour essayer de traiter le problème dans un cas général et pas uniquement sur un exemple.

Question 5

Un grand nombre de copies ne répond pas correctement à cette question, et se contente de paraphraser l'exemple qui précède sur les fonctions de tri. L'énoncé indiquait pourtant que cet exemple était indépendant

du problème étudié ici. Cet exemple était présent uniquement pour donner une indication sur le type de contenu attendu en réponse.

L'élément essentiel à mentionner est la comparaison entre la complexité temporelle de chaque méthode et le nombre de fois où elle est appelée. Les ordres de grandeur précis pourront être donnés en question 8.

Question 6

L'explication des algorithmes vise à séparer la difficulté entre, d'une part, la conception de l'algorithme et, d'autre part, sa rédaction en Python, qui elle n'est demandée qu'en question 7.

L'explication d'un algorithme pose cependant des problèmes à de nombreux candidats, qui soit ne traitent pas la question, soit rédigent des explications imprécises et beaucoup trop longues.

Malgré les précautions prises par le sujet pour diriger les candidats vers cette solution, l'idée du pré-calcul complet de toutes les traversées a rarement été vue. C'est pourtant celle-ci qui garantit le plus court temps d'exécution pour `traverse`.

Question 7

Certaines copies parviennent à une implémentation correcte et efficace, sans avoir réussi à la décrire précisément à la question précédente.

Une grande partie des copies ne répond pas aux contraintes de l'énoncé et propose une solution où `__init__` stocke les portails, parfois en initialisant la grille elle-même en mémoire, et `traverse` effectue une simulation du mouvement du robot, donc en temps non constant.

Le code est plus facile à écrire si on a compris que tous les chemins sont disjoints. Si l'on n'a pas vu ce résultat, il reste envisageable de mémoïser les appels pour un éventuel cas où on repasserait plusieurs fois par le même morceau de chemin : le but de cette optimisation est de garantir, comme demandé par le sujet, que `__init__` est la plus rapide possible, dès lors que l'on a déjà rendu `traverse` en temps constant.

Question 8

Un calcul précis est attendu. Des réponses du type « l'algorithme est quadratique » sans autre précision sont nécessairement incomplètes.

La réponse fournie doit être cohérente avec le code qui a été écrit, ou, si la question 7 n'est pas traitée, avec l'algorithme décrit en question 6. Le jury ne voit pas comment il serait possible d'estimer un temps de calcul quand ni la question 6 ni la question 7 n'ont été traitées.

Question 9

Le corrigé fournit volontairement un plus grand nombre de tests que celui qui était attendu par l'énoncé, afin d'essayer de couvrir largement toutes les idées qui auraient pu être avancées dans les copies.

Le jury regrette que trop peu de candidats abordent correctement cette question, qui permet pourtant de montrer que l'on a un vrai recul sur les notions de validation de code. Proposer des jeux de tests variés, qui visent à couvrir l'intégralité du code potentiellement écrit, est une démarche nécessaire dans la gestion de projets de plus grande ampleur, et il est indispensable que cette démarche soit acquise dans un enseignement d'informatique, en particulier avec une composante pratique.

Le jury a été attentif à ce que les tests proposés par les candidats soient variés, et ne consistent pas en une déclinaison du même test.

Question 10

Cette question est globalement peu traitée, et de nombreux candidats se contentent de répéter des éléments déjà abordés dans les grilles statiques sans vraiment souligner les différences sur les grilles dynamiques.

Certes, les mêmes algorithmes fourniront le bon résultat, mais avec une complexité temporelle moins bonne que ce qui était attendu et demandé par le sujet.

Question 11

Il s'agit d'une question qui évalue fortement les compétences en programmation Python. Les codes attendus peuvent être assez longs : il est impératif de bien structurer son code pour qu'il soit compréhensible.

Question 12

Cette question a été peu abordée, et peu de réponses correctes ont été fournies. Seules quelques excellentes copies parviennent à donner une réponse argumentée et cohérente avec le code de la question précédente.

Question 13

Il s'agit d'une question simple de compréhension, qu'il convient de traiter rapidement et correctement. Le jury a apporté une attention particulière à la justification de chaque résultat, qui était explicitement demandée par le sujet.

Question 14

Cette question a souvent été bien traitée, même si de nombreuses copies ne proposent pas un code qui s'exécute en espace constant. Il n'est pas nécessaire ici de mémoriser quelles variables sont vivantes à chaque instant.

Question 15

Cette question a été majoritairement bien traitée. Il est inutile de chercher une politique d'affectation compliquée pour le choix d'un registre à chaque évènement `début`, une pile suffit par exemple.

Question 16

Les coloriages proposés ont été quasiment tous corrects. En revanche, il ne faut pas négliger la preuve de minimalité pour chaque coloriage.

Question 17

Cette question ne pose pas de difficulté particulière et a été très largement bien traitée.

Question 18

Le code attendu pour cette question doit rester simple. Il est en particulier inutile de se lancer dans l'écriture d'un parcours de graphe, comme cela a pu être vu dans des copies. De plus, on s'attend à ce que l'algorithme utilisé soit linéaire, là où plusieurs copies proposent un algorithme quadratique.

Question 19

Question globalement bien traitée, mais ici encore il ne faut pas négliger la justification.

Question 20

Question de programmation un peu plus technique, qui nécessite de bien comprendre l'algorithme proposé par le sujet et d'être à l'aise avec la programmation en Python pour une mise en œuvre correcte.

Il est impératif de bien structurer son code pour aboutir à une solution assez concise. L'introduction de fonctions auxiliaires est un excellent moyen d'y parvenir, mais dans ce cas, il est indispensable pour chaque fonction introduite d'en donner une spécification claire pour aider à la compréhension.

Question 21

Les optimisations demandées ne sont pas très difficiles mais demandent néanmoins d'avoir bien compris le problème. Cette question a été majoritairement peu traitée, mais les meilleures copies fournissent des réponses pertinentes.

Vu la formulation de la question, il est inutile de programmer les optimisations. Certains candidats, à l'aise en programmation, ont cependant trouvé cette approche plus efficace : ceci est accepté par le jury tant que les deux optimisations sont clairement indiquées dans le code.

Question 22

Question sans difficulté particulière, mais qui est importante pour comprendre que l'algorithme précédent n'est pas correct pour trouver un coloriage dans tous les cas.

Question 23

Question qui a été bien traitée par les meilleures copies.

Question 24

Cette question est difficile et a été très peu traitée. La plupart des réponses données n'étaient pas pertinentes.

Question 25

Cette question dépend totalement de la précédente, et elle a donc été très peu traitée correctement. Le code ne présente pas de difficulté particulière une fois que l'on a compris l'optimisation attendue.

Question 26

Cette question d'introduction a été majoritairement bien traitée par les candidats qui ont abordé cette partie du sujet.

Question 27

L'erreur principale sur cette question a été d'oublier qu'il fallait traverser récursivement l'arbre de syntaxe de l'expression donnée en paramètre.

Question 28

La structure du code attendu est similaire à la question précédente : cette question ajoute simplement la difficulté de gérer une table pour renommer les variables dans les membres droits des affectations.

Question 29

Il y a plusieurs réponses possibles. On peut choisir de mettre des φ à la fin des `if` y compris pour les variables que l'on n'utilise plus ensuite. Les versions qui composent plusieurs φ dans une même expression ne sont pas acceptées, cela ne correspond pas à la définition donnée par le sujet.

Question 30

Question facile, bien traitée par les candidats qui sont parvenus jusqu'à cette partie. Il faut impérativement bien respecter la règle de construction : dans la condition du `while`, le seul φ autorisé doit correspondre à une affectation identique au début du corps de boucle.

Question 31

Question facile avec de très nombreuses réponses possibles.

Question 32

Le jury a apprécié les réponses où était esquissée l'idée d'un tri topologique des composantes fortement connexes.

Question 33

Il est important de rédiger un raisonnement où tous les cas sont envisagés, pour démontrer l'impossibilité d'un tel graphe. Exhiber un exemple en indiquant qu'il n'engendre pas le bon graphe n'est pas une justification suffisante.

Question 34

Question bien réussie, parmi les copies peu nombreuses qui l'ont abordée.

Question 35

Les rares candidats qui ont bien compris cette partie ont réussi à décrire la bonne relation.

Question 36

Question difficile qui a été peu traitée. De rares copies parviennent à esquisser l'argument de terminaison.

Question 37

Question très ouverte et difficile, qui n'a été traitée par aucune copie.

7 Proposition de corrigé

Le jury propose un corrigé de l'épreuve afin d'aider les futurs candidats à préparer leur épreuve. De nombreuses autres réponses étaient possibles et ont été acceptées par le jury.

Question 1a

Le robot passe par les portails 3 (entrée en colonne 1, sortie en colonne 4) puis 1 (entrée en colonne 4, sortie en colonne 6) puis 7 (entrée en colonne 6, sortie en colonne 2). Il atteint la dernière ligne de la grille en colonne 2.

Question 1b

Le robot reste toujours dans la colonne 7 et passe par les portails 5 (ligne 1, téléporté en ligne 3), 9 (ligne 4, téléporté en ligne 2), 5 (ligne 3, téléporté en ligne 1), 9 (ligne 2, téléporté en ligne 4), avant d'atteindre la dernière ligne, en colonne 7.

Question 1c

Si le robot rentre par la colonne 8, il sera téléporté sur le portail 6 de la colonne 2 pour arriver sur le portail 4, qui est unique. Le robot restera donc coincé sur cette case.

Question 2a

`G = GrilleRobot(9, 6, [(4,2,1), (6,1,1), (2,2,4)])` est une instanciation valide. Elle correspond à la grille de la figure 1 avec seulement les portails 1 et 4.

`G = GrilleRobot(9, 6, [(4,2,1), (6,1,1), (9,2,4)])` est une instanciation invalide car la liste contient un portail en position $(9, 2)$ qui est en dehors de la grille.

Question 2b

Avec la grille valide précédente, `G.traverse(0)` est un appel valide, alors que `G.traverse(9)` est un appel invalide car la grille ne contient pas de colonne numéro 9.

Question 3

On peut ajouter un appel à une méthode auxiliaire dans `__init__` :

```
class GrilleRobot:
    def __init__(self,
                 nb_colonnes: int,
                 nb_lignes: int,
                 portails: List[Tuple[int,int,int]]) -> None:
        if DEBUG:
            self.check_init_parameters(nb_colonnes, nb_lignes, portails)
```

La méthode auxiliaire est définie de la façon suivante :

```
    def check_init_parameters(self,
                              nb_colonnes: int,
                              nb_lignes: int,
                              portails: List[Tuple[int,int,int]]) -> None:
        assert 0 < nb_colonnes
        assert 2 <= nb_lignes

        # Dictionnaire pour vérifier qu'on ne met pas 2 portails dans la
        # même case. Il est possible aussi d'utiliser un ensemble.
        cases_occupees = {}
        # Dictionnaire pour compter le nombre de portails de chaque numéro
        nombre_portails: Dict[int,int] = {}
        for x, y, p in portails:
            assert 0 <= x < nb_colonnes
            # attention, pas de portails sur les première / dernière lignes
            assert 0 < y < nb_lignes - 1

            assert 0 <= p < (nb_lignes - 2) * nb_colonnes
            assert (x,y) not in cases_occupees
            cases_occupees[(x, y)] = p    # on peut remplacer `p` par `True`
            nb = nombre_portails.get(p, 0)
            assert nb == 0 or nb == 1
            nombre_portails[p] = nb + 1
```

Question 4

Un robot ne peut **pas** rester bloqué dans la grille, sans jamais atteindre la dernière ligne, sauf s'il passe par un portail unique et y reste coincé.

Supposons que le robot n'arrive jamais sur la dernière ligne mais qu'il ne passe pas par un portail unique. Il doit donc passer par une infinité de portails. Supposons que le robot rentre dans les portails $(x_0, y_0, n_0) \to (x_1, y_1, n_1) \to (x_2, y_2, n_2) \to \cdots$.

Comme la grille est finie, cette liste contient des doublons : le robot rentre nécessairement à nouveau dans un portail sur lequel il est déjà rentré. Notons (x_k, y_k, n_k) le *premier* portail répété dans cette suite, c'est-à-dire celui avec k minimal : $(x_0, y_0, n_0) \to \cdots \to \underline{(x_k, y_k, n_k)} \to \ldots \to (x_{k+\ell}, y_{k+\ell}, n_{k+\ell}) \to \underline{(x_k, y_k, n_k)} \to \ldots$.

— Il ne peut pas s'agir du premier portail rencontré dans la colonne initiale ($k = 0$) : en effet, le premier portail rencontré est forcément le plus haut dans la colonne de départ. Pour que le robot repasse par ce portail, il faudrait que le portail $(x_{k+\ell}, y_{k+\ell}, n_{k+\ell})$ renvoie le robot au-dessus de (x_0, y_0, n_0), ce qui est impossible.
— S'il ne s'agit pas du premier portail rencontré, on a alors deux portails qui envoient le robot sur (x_k, y_k, n_k) :

  — $(x_{k-1}, y_{k-1}, n_{k-1})$
  — et $(x_{k+\ell}, y_{k+\ell}, n_{k+\ell})$.

  Ces deux portails sont distincts, sinon $k-1$ donnerait un portail répété avant k. La colonne x_k doit contenir des portails $n_{k+\ell}$ et n_{k-1} au-dessus du portail (x_k, y_k, n_k). Par exemple, un portail $n_{k+\ell}$ au-dessus d'un portail n_{k-1} au-dessus du portail n_k. Mais dans ce cas, $(x_{k+\ell}, y_{k+\ell}, n_{k+\ell})$ envoie le robot sur n_{k-1} au lieu de n_k !

  Le raisonnement est similaire si n_{k-1} est au-dessus de $n_{k+\ell}$ qui est au-dessus de n_k.

Dans tous les cas, nous obtenons une contradiction. Un robot qui ne passe jamais par un portail unique ne peut pas rester dans la grille sans atteindre la dernière ligne.

Question 5

Si l'utilisation attendue de la classe `GrilleRobot` est de créer une grille et de faire de nombreux appels à la méthode `traverse` (par exemple dans une boucle), la complexité de `__init__` n'aura pas beaucoup d'impact sur la durée d'exécution du programme entier. Le temps passé à exécuter la méthode `__init__` sera négligeable devant tous les appels à `traverse`.

Par contre, dans le cas où l'on initialise de nombreuses grilles différentes et pour chacune d'elles on appelle une seule fois la méthode `traverse`, le temps d'exécution de `__init__` aura un impact beaucoup plus important sur le temps d'exécution du programme complet.

Question 6

Il y a plusieurs points importants si l'on souhaite optimiser la complexité de `traverse` :

— la méthode `traverse` est une fonction finie : elle doit renvoyer un entier dans $\{-1, 0, 1, \ldots, n-1\}$ pour chaque entier dans $\{0, 1, \ldots, n-1\}$ (n est ici le nombre de colonnes). On peut donc *précalculer* tous les résultats possibles pour `traverse` dans `__init__` et les stocker dans un tableau de taille n. La méthode `traverse` répondra alors en temps constant ;
— pour ce précalcul, on simule simplement les déplacements du robot. Comme l'énoncé se place dans le cas où la grille est pleine, il n'est pas nécessaire d'essayer d'optimiser la descente du robot en détectant plusieurs cases vides successives lorsqu'il descend sur une même colonne ;
— il est utile d'avoir une structure qui à chaque coordonnée de case dans la grille associe l'emplacement de l'autre portail, afin de ne pas payer le coût en temps de recherche de l'autre portail lors du précalcul d'un déplacement de robot. On peut utiliser pour cela un dictionnaire, ou bien la grille elle-même, ce qui a le même coût car la grille est totalement remplie ici.

Question 7

Pour la méthode `__init__`, on utilise une méthode auxiliaire `traverse_aux` qui effectue un mouvement de robot depuis une case donnée.

```
def __init__(self,
             nb_colonnes: int,
             nb_lignes: int,
             portails: List[Tuple[int,int,int]]) -> None:
```

```
    if DEBUG:
        self.check_init_parameters(nb_colonnes, nb_lignes, portails)

    self.nb_colonnes = nb_colonnes
    self.nb_lignes = nb_lignes
    self.portails = portails[:]

    # G contient un tableau à 2 dimension contenant, pour chaque case :
    #   - None lorsque la case est vide
    #   - une paire (n, ...) lorsqu'elle contient un portail de numéro n
    # Le second composant de la paire est
    #   - None lorsque ce portail est unique
    #   - une paire (x,y) lorsque le second portail de même numéro se
    #   trouve dans la case (x,y)
    self.G = [[None]*nb_colonnes for _ in range(nb_lignes)]

    # O contient, pour chaque numéro de portail, la liste des coordonnées
    # de cases le contenant. (Les listes sont donc de taille 1 ou 2.)
    O: Dict[int, List[Tuple[int,int]]] = {}

    # initialisation de G, en 2 temps.
    # d'abord, les numéros de portails
    for x,y,p in portails:
        self.G[y][x] = (p, None)
        O[p] = O.get(p, [])      # on enregistre les coordonnées du portail n° n
        O[p].append((x,y))       # pour les retrouver rapidement plus tard

    # ensuite, les coordonnées du portail jumeau
    for p, L in O.items():
        if len(L) == 1:
            continue
        x1, y1 = L[0]
        x2, y2 = L[1]
        self.G[y1][x1] = (p, (x2,y2))
        self.G[y2][x2] = (p, (x1,y1))

    # tableau précalculé des colonnes de sortie
    self.Sortie = [-2] * nb_colonnes


    # version 1 : on sait que les chemins sont disjoint, on peut
    # utiliser une méthode traverse_aux naive.
    # on appelle la méthode traverse_aux sur chaque case de la première
    # ligne pour remplir le tableau sortie
    for x in range(nb_colonnes):
        self.Sortie[x] = self.traverse_aux(x)

def traverse_aux(self, x: int) -> int:
    y = 0
    while y != self.nb_lignes-1:
        y += 1          # le robot descend d'une case
        portail = self.G[y][x]
        if portail is not None:
            # le robot est arrivé sur une case contenant un portail
            coord_autre_portail = portail[1]
            if coord_autre_portail is None:
```

```
                # il s'agit d'un portail unique, on est coincé !
                return -1
            # le robot est téléporté sur le portail jumeau
            x, y = coord_autre_portail
    return x
```

```
def traverse_aux_mem(self, x: int, y: int, M: List[List[Optional[int]]]) -> int:

    r = M[y][x]
    if r is not None:
        # on a déjà calculé cette valeur, on la réutilise
        return r

    if y == self.nb_lignes - 1:
        r = x
    else:
        sy = y + 1      # le robot descend d'une case

        portail = self.G[sy][x]
        if portail is None:
            # il n'y a pas de portail, on continue normalement
            r = self.traverse_aux_mem(x, sy, M)
        else:
            # il y a un portail
            coord_autre_portail = portail[1]
            if coord_autre_portail is None:
                # ce portail unique, on arrête tout, on est coincé !
                r = -1
            else:
                # ce portail est double, on se téléporte
                tx, ty = coord_autre_portail
                r = self.traverse_aux_mem(tx, ty, M)
    M[y][x] = r
    return r
```

La méthode `traverse` est alors en temps constant :

```
def traverse(self, colonne: int) -> int:
    return self.Sortie[colonne]
```

Question 8

Les chemins potentiellement empruntés par le robot sont tous disjoints. En effet, le même raisonnement fait dans la réponse à la question 4 montre qu'avec deux points de départ différents sur la première ligne, le robot ne peut pas *rentrer* dans deux portails identiques.

Au pire cas, les `nb_colonnes` appels à `traverse_aux` vont passer par toutes les cases `(x,y)` de la grille. Comme les chemins sont tous disjoints par la remarque précédente, ces appels ne peuvent pas passer deux fois par la même case.

Cela donne un temps de calcul de l'ordre de `nb_colonnes*nb_lignes` au pire cas pour `__init__`, et un temps constant pour `traverse`. La mémoire utilisée est de l'ordre de `nb_colonnes*nb_lignes`, pour stocker la grille.

Un exemple de grille atteignant cette complexité dans la situation considérée serait une grille de la forme

	0	1	2	3	4	5	6	7
0								
1	**0**	**3**	**6**	···	···	···	···	···
2	**0**	**3**	**6**	···	···	···	···	···
3	**1**	**4**	**7**	···	···	···	···	···
4	**1**	**4**	**7**	···	···	···	···	···
5	**2**	**5**	**8**	···	···	···	···	···
6	**2**	**5**	**8**	···	···	···	···	···
7								

Question 9

Les bugs les plus probables sont les bugs sur les cas limites : valeur la plus petite ou la plus grande possible d'un paramètre ou d'une propriété. Il est aussi fréquent d'avoir des bugs où la première itération d'un processus fonctionne, mais pas la suivante. Un autre type de bug courant consiste en des bugs où une supposition sur les données d'entrée a été effectuée, qui n'est pas imposée par le sujet. Un autre bug typique est une inversion entre les deux coordonnées.

Voici une liste de situations à considérer dans nos tests :

- grille avec un portail unique,
- grille avec un portail double,
- grille avec une colonne vide,
- grille ne contenant aucun portail : cas limite,
- grille de deux lignes ou d'une seule colonne : cas limites, plus petites tailles possibles,
- grille non carrée : détection d'inversions entre coordonnées x et y,
- grille avec des portails sur la première et dernière colonne et sur la deuxième et l'avant-dernière ligne : cas limites, plus petites et plus grandes coordonnées de portails possibles,
- portails consécutifs (l'un au-dessus de l'autre),
- valeur extrême de numéro de portail : 0 et `nb_colonnes*(nb_lignes-2)-1`,
- cas où les numéros des portails ne sont pas consécutifs,
- cas où un parcours passe par plusieurs paires de portails,
- cas où on passe 2 fois par la même paire de portails.

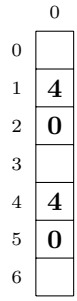
On peut également tester que l'ordre des portails donnés à `__init__` ne modifie pas le résultat.

Voici des exemples de grilles :

- Grille sans aucun portail, avec seulement deux lignes

	0	1	2
0			
1			

  avec comme résultat attendu : `[0, 1, 2]`
- Grille avec une seule colonne (donc non carrée), des portails sur la deuxième et l'avant-dernière ligne, des portails l'un au-dessus de l'autre, des portails de numéros non consécutifs, des portails numérotés `0` et `nb_colonnes*(nb_lignes-2)-1` et un cas où le parcours passe par plusieurs portails, et deux fois par la même paire de portails :

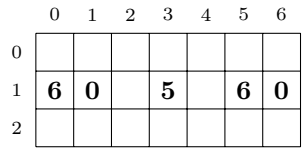


avec comme résultat attendu : `[0]`

On remarque que comme il n'y a qu'une seule colonne, un tel test ne donnera que peu d'indication sur le fait que le parcours effectué soit le bon, mais teste surtout que la situation ne cause pas de crash ou de boucle infinie.

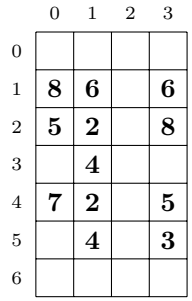
On peut tester en donnant la liste des portails dans l'ordre 0, 4, 0, 4 pour tester le cas où les portails de même type ne sont pas fournis l'un juste après l'autre.

— Grille non carrée de hauteur minimale avec des portails sur la première et dernière colonne, dont des portails doubles et simples, et des colonnes vides, valeur extrême de numéro de portail :



avec comme résultat attendu : `[5, 6, 2, -1, 4, 0, 1]`

— Grille non carrée avec un parcours qui passe par plus de deux portails différents, avec un cas où le 2e portail est un portail unique, et des cas où le portail suivant est juste en dessous, portails sur la deuxième et l'avant-dernière ligne, première et dernière colonne, cas avec une colonne vide.



avec comme résultat attendu : `[-1, -1, 2, 1]`.

Question 10

On conserve, pour chaque colonne, une liste chainée des portails de cette colonne, du plus haut au plus bas. On utilisera pour ceci une classe `Portail` qui contient juste des attributs :

— `x` et `y` pour la position du portail,
— `p` pour le numéro du portail,
— `suivant` pour le portail suivant.

La méthode `__init__` initialise ces listes : ordre `nb_colonnes`, et un dictionnaire `D` vide : temps constant. Ce dictionnaire servira à trouver rapidement le jumeau d'un portail : `D[p]` donnera la liste des portails avec le numéro `p`.

La méthode `ajoute` insère un nouveau portail dans une liste. Il faut parcourir la liste pour trouver où l'insertion doit être faite (ordre `min(nb_lignes, nb_portails)`) puis faire l'insertion (temps constant). Elle ajoute également les coordonnées du nouveau portail dans le dictionnaire `D` (temps constant).

La méthode `traverse` fonctionne de manière similaire à la méthode `traverse_aux` de la partie précédente, mais au lieu de descendre d'une case à chaque fois, elle descend directement au portail suivant dans la colonne, en utilisant la liste chainée (ordre `min(nb_lignes, nb_portails)`).

À chaque portail, on utilise le dictionnaire `D` pour chercher le portail jumeau et continuer le parcours, ou s'arrêter s'il s'agit d'un portail unique (temps constant).

Question 11

On utilise une classe `Portail` pour stocker la liste des portails.

```
class Portail:
    x: int                          # coordonnées du
    y: int                          # portail dans la grille
    p: int                          # numéro du portail
    suivant: Optional["Portail"]    # portail suivant dans la liste

    def __init__(self, x: int, y: int, p: int,
                 suivant: Optional["Portail"] = None) -> None:
        self.x = x
        self.y = y
        self.p = p
        self.suivant = suivant
```

```
    def __init__(self, nb_colonnes: int, nb_lignes: int) -> None:

        self.nb_colonnes = nb_colonnes
        self.nb_lignes = nb_lignes

        # listes chainées des portails dans chaque colonne
        # initialement, toutes les listes sont vides (None)
        self.Portails: List[Optional[Portail]]
        self.Portails = [None] * nb_colonnes

        # dictionnaire contenant les différents portails ajoutés, indexés par
        # leur numéro
        self.D: Dict[int,List[Portail]]
        self.D = {}
```

```
    def ajoute_portail(self, x: int, y: int, p: int) -> None:

        P = Portail(x, y, p)
        # on ajoute le nouveau portail dans self.D
        if self.D.get(p) is None:
            self.D[p] = [P]
        else:
            # on pourrait valider les arguments ici
            self.D[p].append(P)

        # insertion de P dans la liste chainée self.Portails[x]
        L: Optional[Portail] = self.Portails[x]
        if L is None:
            # si la liste était vide
            self.Portails[x] = P
        elif L.y > y:
            # le premier élément de la liste est plus bas que le nouveau
            # portail: on l'insère donc en tête de liste
            P.suivant = L
```

```
            self.Portails[x] = P
        else:
            # on cherche le portail juste avant P dans L
            prec = L
            while prec.suivant is not None and prec.suivant.y < y:
                prec = prec.suivant
            # on insère P juste après prec
            P.suivant = prec.suivant
            prec.suivant = P

    def traverse(self, x: int) -> Tuple[int,List[int]]:
        portail_courant = self.Portails[x]
        portails_vus = []

        while portail_courant is not None:
            p = portail_courant.p
            portails_vus.append(p)
            if len(self.D[p]) == 1:
                # s'il n'y a qu'un seul portail numero p,
                # il doit s'agir du portail courant
                # et on est donc coincé
                return -1, portails_vus
            else:
                assert len(self.D[p]) == 2
                P1, P2 = self.D[p]
                if P1.x == portail_courant.x and P1.y == portail_courant.y:
                    # si le portail courant est en premier dans self.D[p],
                    # son jumeau est en second. C'est là qu'on se téléporte.
                    portail_courant = P2
                else:
                    # sinon, son jumeau est en premier, et portail_courant est
                    # en second
                    assert P2.x == portail_courant.x and P2.y == portail_courant.y
                    portail_courant = P1
                x = portail_courant.x                       # on met la colonne à jours
                portail_courant = portail_courant.suivant   # et on descend jusqu'au portail suivant

        return x, portails_vus
```

Question 12

Dans la situation décrite, `nb_portails` est inférieur à `nb_colonnes` et on a donc :

- initialisation : ordre n
- premier appel à `ajoute` puis `traverse`
- …
- appel numéro k à `ajoute` : ordre k, suivi par l'appel numéro k à `traverse` : ordre k
- …

pour un temps total d'exécution d'ordre $n + 2 \times (1 + 2 + \cdots + n) = n \times (n + 2)$, soit ordre n^2.

Question 13

Pour le cas 3, on affecte a au registre R_0, b au registre R_1, c au registre R_2 et d au registre R_3. Cette affectation occupe 4 registres. Après l'instruction « début d », les quatre variables a, b, c et d sont utilisées donc il n'existe pas d'affectation avec moins de registres.

Pour le cas 4, après l'évènement « début c », trois variables sont utilisées donc il faut au moins 3 registres

à cet instant du programme pour les stocker toutes. On atteint exactement 3 registres en affectant a au registre R_0, b au registre R_1, c au registre R_2 et d au registre R_1. Ceci est possible car l'instruction « fin b » est placée avant « début d », donc le registre R_1 est libre au moment de la création de d.

Pour le cas 5, après la ligne « début b », il y a deux variables utilisées, a et b, donc il faut au moins 2 registres. On atteint ce nombre en affectant a au registre R_0, b au registre R_1, c au registre R_0 (ce qui est possible car a est détruite juste avant la création de c) et d au registre R_1 (ce qui est possible car d est créée juste après la destruction de b).

Le cas 6 est similaire au cas 3, avec la même affectation.

Question 14

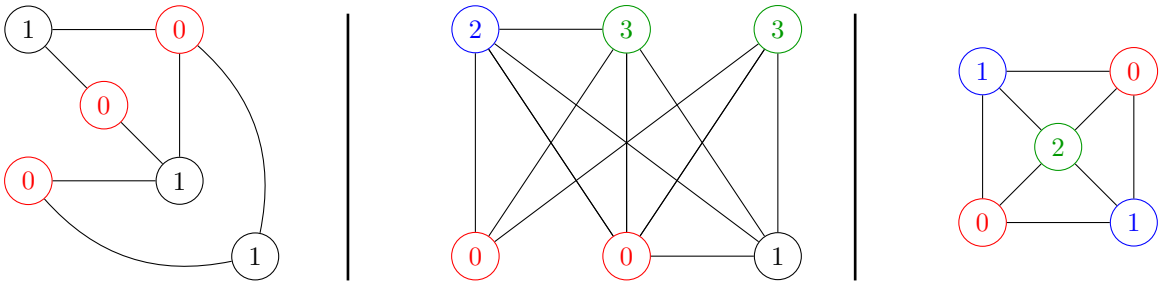
```
def nombre_de_registres_necessaires(n, E):
    nb_variables_utilisees = 0
    res = 0
    for evenement in E:
        if evenement[0] == "fin":
            nb_variables_utilisees -= 1
        else:
            nb_variables_utilisees += 1
        res = max(res,nb_variables_utilisees)
    return res
```

Question 15

```
def affectation_registres(n, E):
    reponse = [-1 for i in range(n)]
    variables_libres = list(range(nombre_de_registres_necessaires(n, E)))
    for evenement in E:
        if evenement[0] == "fin":
            variables_libres.append(reponse[evenement[1]])
        else:
            reponse[evenement[1]] = variables_libres.pop()
    return reponse
```

Question 16

Voici un coloriage de chacun des graphes.



Ces coloriages sont corrects (on peut le voir sur le dessin) et minimaux :

- à gauche il faut au moins deux couleurs,
- au milieu on peut remarquer qu'il y a quatre nœuds qui sont connectés deux à deux (les nœuds de couleurs 1, 2 et leurs voisins de couleurs 0 et 3),
- à droite, le nœud colorié 2 ne peut pas être de la couleur des autres nœuds et il faut au moins deux couleurs pour les autres nœuds.

Question 17

```
def calcul_degre(G):
    return [len(e) for e in G]
```

Question 18

```
def est_valide(G, C):
    for couleur in C:
        if couleur < 0:
            return False
    for u in range(len(G)):
        for v in G[u]:
            if C[u] == C[v]:
                return False
    return True
```

Question 19

Dans le graphe de gauche, si $K \leq 2$, alors le processus bloque immédiatement. Si $K = 3$ alors on peut retirer les sommets dans l'ordre A, B, C, D, E, F et colorier dans le sens inverse ce qui donne F=0, E=0, D=1, C=2, B=2, A=1.

Pour le graphe du milieu, le plus petit degré est 3, on doit donc avoir $K = 4$. On peut alors retirer les sommets dans l'ordre B, A, C, D, E, F ce qui donne comme coloriage : F=0, E=1, D=2, C=1, A=3, B=2.

Pour le graphe de droite, le plus petit degré est 3 (le nœud E est de degré 4), il faut donc $K = 4$ et on peut retirer les nœuds dans l'ordre A, B, C, D, E, ce qui donne E=0, D=1, C=2, B=1, A=2.

Question 20

```
def colorie_noeud(G, coloriage, noeud):
    """
    Fonction auxiliaire qui colorie un nœud en prenant une couleur disponible
    parmi celles non encore utilisées parmi les voisins de `noeud`. Elle
    modifie en place `coloriage`. Les nœuds dont le coloriage vaut `None` ne
    sont pas pris en compte.
    """
    couleurs_prises = {}
    for voisin in G[noeud]:
        if coloriage[voisin] is not None:
            couleurs_prises[coloriage[voisin]] = True

    couleur = 0
    while True:
        if couleur not in couleurs_prises:
            coloriage[noeud] = couleur
            return
        couleur = couleur + 1

def coloriage_heuristique_recursif(G, K, retire, nb_restants):
    n = len(G)
    if nb_restants == 0:
        return [None for _ in range(n)]
    for noeud in range(n):
        if not retire[noeud]:
            # Calcul du degré de `noeud` en ne prenant en compte que les
            # sommets encore présents.
```

```
            deg = 0
            for voisin in G[noeud]:
                if not retire[voisin]:
                    deg +=1
            # Retrait de `noeud` pour le coloriage récursif
            if deg < K:
                retire[noeud] = True
                coloriage = coloriage_heuristique_recursif(G, K, retire, nb_restants-1)
                colorie_noeud(G, coloriage, noeud)
                return coloriage
    erreur_coloriage()

def coloriage_heuristique(G, K):
    n = len(G)
    retire = [False for _ in range(n)]
    return coloriage_heuristique_recursif(G, K, retire, n)
```

Question 21

Pour éviter de recalculer les degrés à chaque retrait d'un nœud, on peut calculer initialement tous les degrés au début de l'algorithme et les stocker dans un tableau. Lorsque l'on retire ensuite un nœud, sa liste de voisins permet de diminuer de 1 le degré de chaque voisin restant.

L'algorithme peut par ailleurs stocker la liste des sommets de degré inférieur à K. On considère comme *à traiter* les nœuds de cette liste. À chaque étape, on regarde le nœud suivant à traiter : si, en le retirant, l'un de ses voisins a un degré qui passe en dessous de K, alors on ajoute ce voisin à la fin des sommets à traiter. On ne retire jamais un sommet de la liste des sommets à traiter : le sommet en cours de traitement est identifié par son indice, croissant au fil de l'exécution, dans la liste des sommets à traiter. On remarque alors que lorsque l'on a traité tous les sommets, les sommets sont placés dans la liste dans l'ordre *inverse* de celui dans lequel ils doivent être coloriés. On effectue alors le coloriage en respectant cet ordre.

```
def coloriage_heuristique_optimise(G, K):
    n = len(G)
    degre = calcul_degre(G)

    # Calcul des sommets initialement de degré inférieur à K
    a_traiter = []
    for i in range(n):
        if degre[i] < K:
            a_traiter.append(i)

    # Traitement des sommets les uns après les autres
    for i in range(n):
        # Si on n'a plus de sommet à traiter alors que l'on n'a pas traité tous
        # les sommets du graphe (car i < n), alors le coloriage a échoué.
        if len(a_traiter) <= i:
            erreur_coloriage()

        # Cette étape peut rendre négatifs les degrés de sommets déjà traités,
        # ce qui n'est pas important car ils ne sont plus considérés par la
        # suite.
        for voisin in G[a_traiter[i]]:
            degre[voisin] -= 1

            # Un voisin n'est ajouté qu'une seule fois dans a_traiter : à la
            # première diminution de son degré qui fait passer le degré sous la
            # valeur K, pas les fois suivantes. Le test "< K" serait faux.
```

```
            if degre[voisin] == K-1:
                a_traiter.append(voisin)

    # Coloriage des nœuds dans le sens contraire de a_traiter
    coloriage = [None for _ in range(n)]
    for i in range(len(a_traiter) - 1, -1, -1):
        colorie_noeud(G, coloriage, a_traiter[i])
    return coloriage
```

Question 22

Le graphe de gauche de la question 16 est coloriable avec 2 couleurs mais le coloriage heuristique bloque avec $K = 2$ car tous les nœuds ont un degré supérieur ou égal à 2.

Question 23

```
def coloriage_lent(G, K):
    valides = []

    def finir_coloriage(i, coloriage):
        """
        Fonction auxiliaire récursive qui va tenter de colorier le nœud
        d'indice `i`, en étendant le coloriage avec toutes les couleurs
        existantes.

        Elle ajoute un coloriage à la liste `valides` définie plus haut
        lorsque tous les sommets ont été correctement coloriés.
        """
        if i == len(G):
            if est_valide(G, coloriage):
                valides.append(coloriage.copy())
        else:
            for couleur in range(K):
                coloriage[i] = couleur
                finir_coloriage(i+1, coloriage)

    finir_coloriage(0, [None for _ in range(len(G))])
    return valides
```

Question 24

On peut mémoriser à chaque instant, lors de la création d'un coloriage partiel, le plus grand numéro de couleur déjà attribué. Le sommet suivant à traiter peut être colorié soit avec une des couleurs déjà utilisées ailleurs dans le graphe, soit uniquement avec la couleur suivante.

Question 25

```
def coloriage_moins_lent(G, K):
    n = len(G)
    coloriage = [None for _ in range(n)]

    def finir_coloriage(i, nb_couleurs):
        """
        Fonction auxiliaire récursive qui tente de colorier les nœuds d'indice
        `i` et suivants, sachant que les nœuds d'indice inférieur à `i` sont
        déjà coloriés avec les couleurs 0 à `nb_couleurs`-1. Elle renvoie True
```

```
        lorsqu'elle a réussi à compléter le coloriage.
        """
        if i == n:
            return True

        # Optimisation sur les coloriages symétriques : parmi les couleurs non
        # encore utilisées, on n'essaie que la couleur `nb_couleurs`.
        couleur_dispo = [True for _ in range(min(nb_couleurs + 1, K))]
        # Optimisation sur les coloriages invalides : on ne considère pas les
        # couleurs des voisins déjà coloriés.
        for j in G[i]:
            if coloriage[j] is not None and coloriage[j] < len(couleur_dispo):
                couleur_dispo[coloriage[j]] = False

        for couleur in range(len(couleur_dispo)):
            if couleur_dispo[couleur]:
                coloriage[i] = couleur
                if finir_coloriage(i+1, max(couleur + 1, nb_couleurs)):
                    return True

        # Remettre la couleur de i à None pour éviter que les autres appels
        # ne considèrent la couleur comme non disponible, car on travaille
        # en place dans le même tableau `coloriage`.
        coloriage[i] = None
        return False

    if finir_coloriage(0, 0):
        return coloriage
    return None
```

Question 26

```
def est_ssa(instructions):
    vues = {}
    for instr in instructions:
        if instr[0] in vues:
            return False
        vues[instr[0]] = True
    return True
```

Question 27

```
def variables_existent(expression, definies):
    """
    Fonction intermédiaire qui prend en paramètre une expression et une liste
    de variables qui sont définies, et renvoie True exactement lorsque toutes
    les variables utilisées dans expression sont présentes dans definies.
    """
    if isinstance(expression, int):
        return True
    elif isinstance(expression, str):
        return expression in definies
    else:
        # Cas d'un opérateur binaire, on ignore la première case et on récupère
        # la liste des variables dans les cases suivantes.
        return variables_existent(expression[1], definies) and \
```

```
                variables_existent(expression[2], definies)
def affectations_valides(instructions):
    definies = {}
    for instr in instructions:
        if not variables_existent(instr[1], definies):
            return False
        definies[instr[0]] = True
    return True
```

Question 28

À chaque affectation, on renomme la variable modifiée en lui ajoutant un numéro qui correspond au numéro de l'affectation depuis le début du programme. Il faut ensuite propager le renommage aux expressions suivantes dans le programme : on le fait avec un dictionnaire qui à chaque nom de variable d'origine associe le nouveau nom.

```
def renomme_variables(expression, renomme):
    """
    Fonction intermédiaire qui renomme toutes les variables présentes dans
    "expression" en utilisant le dictionnaire "renomme" qui à chaque ancien nom
    associe son nouveau nom.
    """
    if isinstance(expression, int):
        return expression
    if isinstance(expression, str):
        return renomme[expression]
    return (
        expression[0],
        renomme_variables(expression[1], renomme),
        renomme_variables(expression[2], renomme)
    )

def simple_vers_ssa(instructions):
    # Compteur qui permet de créer de nouvelles variables jamais vues avant.
    compteur = 0
    # Dictionnaire qui indique comment renommer les variables, la clé est
    # l'ancien nom et la valeur le nouveau nom.
    renomme = {}

    res = []
    for instr in instructions:
        # On crée une nouvelle variable pour remplacer celle qui est donnée
        # à gauche de l'affectation dans "instr".
        nouveau_nom = f"{instr[0]}{compteur}"
        res.append((nouveau_nom, renomme_variables(instr[1], renomme)))
        renomme[instr[0]] = nouveau_nom
        compteur += 1

    return res
```

Question 29

```
x0 = A
y0 = B
r0 = x0 % y0
fin0 = False
```

```
if r0 == 0:
    fin1 = True
else:
    x1 = y0
    y1 = r0
    r1 = x1 % y1
    if r1 == 0:
        fin2 = True
    else:
        x2 = y1
        y2 = r1
        r2 = x2 % y2
    y3 = phi(y1, y2)
y4 = phi(y0, y3)
res0 = y4
```

Question 30

```
n0 = 42
x0 = 0
y0 = 16
while phi(x0, x2) < n0:
    x1 = phi(x0, x2)
    y1 = phi(y0, y2)
    y2 = y1 + x1
    x2 = x1 + 1
x3 = phi(x0, x2)
y3 = phi(y0, y2)
x4 = x3 * y3
```

Question 31

On peut proposer par exemple n'importe quel code où une variable n'est pas modifiée à l'intérieur du branchement ou de la boucle.

Question 32

On propose :

```
while ...:
  while ...:
    ...
  if ...:
    if ...:
      ...
    else:
      ...
  else:
    ...
  ...
```

Question 33

Non, un tel programme n'existe pas.

Le bloc B_2 a deux arêtes sortantes (vers B_0 et B_3) : c'est donc la condition d'un `if` ou d'un `while`. Il a aussi deux arêtes entrantes (depuis B_1 et B_4). Or la condition d'un `if` n'a qu'un seul prédécesseur,

puisqu'on place toujours un bloc de base après un `if` pour rejoindre ses deux branches. B_2 est donc l'en-tête d'un `while` : l'une de ses arêtes entrantes vient du code avant la boucle, l'autre de la fin du corps de boucle.

— Si B_4 termine le corps de boucle, B_4 doit être atteignable depuis B_2, ce qui est impossible car B_4 n'a aucune arête entrante.
— Sinon, B_1 termine le corps de boucle, qui commence en B_0 ou en B_3, et l'autre bloc suit la boucle. Si le corps commence en B_3, alors B_0, qui est après la boucle, pointe vers B_1, au milieu du corps de boucle. Si le corps commence en B_0, alors B_3, qui est après la boucle, pointe vers B_1, au milieu du corps de boucle. Dans les deux cas, on revient dans le corps de la boucle sans repasser par son en-tête, ce qui est impossible avec des `if` et des `while`.

Question 34

— Pour le bloc 0, l'ensemble dominant est $\{0\}$.
— Pour le bloc 1, l'ensemble dominant est $\{0, 1\}$.
— Pour le bloc 2, l'ensemble dominant est $\{0, 8, 7, 3, 2\}$.
— Pour le bloc 3, l'ensemble dominant est $\{0, 8, 7, 3\}$.
— Pour le bloc 4, l'ensemble dominant est $\{0, 8, 7, 3, 4\}$.
— Pour le bloc 5, l'ensemble dominant est $\{0, 8, 7, 5\}$.
— Pour le bloc 6, l'ensemble dominant est $\{0, 8, 7, 6\}$.
— Pour le bloc 7, l'ensemble dominant est $\{0, 8, 7\}$.
— Pour le bloc 8, l'ensemble dominant est $\{0, 8\}$.
— Pour le bloc 9, l'ensemble dominant est $\{0, 8, 9\}$.
— Pour le bloc 10, l'ensemble dominant est $\{0, 8, 7, 3, 10\}$.

Question 35

On a l'égalité $\mathrm{Dom}(B) = \{B\} \cup \bigcap_{B' \in \mathrm{pred}(B)} \mathrm{Dom}(B')$.

Question 36

On initialise :

— $\mathrm{Dom}(B_{\mathrm{debut}}) = \{B_{\mathrm{debut}}\}$,
— et pour tous les autres blocs B, $\mathrm{Dom}(B)$ est initialement l'ensemble de tous les blocs.

On utilise la relation de la question précédente pour mettre à jour ces ensembles jusqu'à ce qu'aucun d'entre eux ne change plus.

D'une itération à l'autre, chaque ensemble $\mathrm{Dom}(B)$ est décroissant. Or, ils sont tous finis donc il existe une itération où tous les dominants sont stables.

```
def blocs_dominants(pred):
    n = len(pred)

    # On suppose que B_debut est B_0.
    B_debut = 0

    dom = [set(range(n)) for _ in range(n)]
    dom[B_debut] = {B_debut}

    changement = True
    while changement:
        changement = False
        for i in range(n):
            if i == B_debut:
                continue
```

```
            inter = None
            for p in pred[i]:
                if inter is None:
                    inter = dom[p].copy()
                else:
                    inter.intersection_update(dom[p])

            if inter is None:
                nouveau_dom = {i}
            else:
                nouveau_dom = inter.union({i})

            if dom[i] != nouveau_dom:
                dom[i] = nouveau_dom
                changement = True

    return [list(d) for d in dom]
```

Question 37

```
def vers_ssa_dominance(blocs, suivants):
    n = len(blocs)
    pred = [[] for _ in range(n)]
    for i in range(n):
        for s in suivants[i]:
            pred[s].append(i)

    dom = blocs_dominants(pred)

    # Calcul de la frontière de dominance simplifiée (DF)
    df = [set() for _ in range(n)]
    for i in range(n):
        if len(pred[i]) >= 2:
            for p in pred[i]:
                courant = p
                while courant != i and (i not in dom[courant] or courant == i):
                    df[courant].add(i)
                    # Simplification : on remonte vers le dominateur immédiat (idom)
                    # Cette étape nécessite logiquement le calcul strict de l'idom
                    idom = [d for d in dom[courant] if d != courant]
                    if not idom:
                        break
                    courant = idom[-1]

    # Copie et ajout des fonctions phi
    nouveaux_blocs = [list(b) for b in blocs]

    # Dictionnaire inversé: variable -> set(blocs où elle est définie)
    defs = {}
    for i, b in enumerate(blocs):
        for var in b:
            if var not in defs:
                defs[var] = set()
            defs[var].add(i)
```

```
for var, blocs_def in defs.items():
    a_traiter = list(blocs_def)
    traites = set()
    while a_traiter:
        b = a_traiter.pop()
        for frontiere in df[b]:
            if frontiere not in traites:
                instruction_phi = [var, var, var]
                nouveaux_blocs[frontiere].append(instruction_phi)
                traites.add(frontiere)
                if frontiere not in defs[var]:
                    defs[var].add(frontiere)
                    a_traiter.append(frontiere)

# L'étape finale de renommage avec incrémentation des compteurs n'est pas
# détaillée ici par concision, mais impliquerait un parcours en profondeur
# de l'arbre des dominateurs.
return nouveaux_blocs
```