CONCOURS GÉNÉRAL DES LYCÉES

—

SESSION 2025

—

Numérique et sciences informatiques

RAPPORT DE JURY

Rapport du jury de concours général des lycées

Épreuve de numérique et sciences informatiques

Session 2025

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 de Savoie Mont Blanc, vice-président du jury.
— 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.
— David Roche, professeur, académie de Grenoble.
— Laurent Sartre, professeur, académie de Bordeaux.
— Amélie Stainer, professeure, académie de Rennes.

2 Le rôle du concours général des lycées dans l'enseignement de NSI

La mise en place du concours général des lycées dans l'enseignement de NSI n'ayant pas été concomitante avec la mise en place de cette spécialité, il semble important au jury de rappeler le rôle structurant qu'un tel concours peut avoir dans les enseignements.

De nombreuses initiatives d'excellence existent en informatique. Certaines étant très anciennes, comme les Olympiades Internationales d'Informatique ou certains concours. D'autres, plus récentes, comme les Trophées NSI ou les Olympiades Académiques de NSI. Le concours général doit être vu comme un complément à celles-ci. Sa forme plus traditionnelle d'épreuve individuelle, reposant sur un sujet à traiter dans un temps limité sous format papier, permet d'offrir aux candidats une autre facette de l'informatique.

Tout en restant dans les limites induites par le programme officiel, le concours général permet, en visant l'excellence dans la maîtrise de l'ensemble des notions, de faire vivre ce programme d'une manière très différente. Ce faisant, sa préparation est très structurante pour les meilleurs élèves d'une classe. En visant une pleine compréhension des notions, les élèves peuvent ainsi commencer à envisager la discipline sous un angle qui pourra être suivi dans les études supérieures, notamment dans les classes préparatoires MP2I/MPI. Cette logique se retrouve dans les sujets du concours général dont la forme est parfois proche des sujets d'informatique des études supérieures.

Les sujets proposés pour l'épreuve du concours général peuvent être exploités dans le cadre de l'enseignement de NSI. Pour faciliter leur réutilisation, le jury met à disposition le corrigé de cette épreuve, accompagné de quelques ressources complémentaires qui prolongent ce sujet, sur la Forge des communs numériques éducatifs, à l'adresse https://forge.apps.education.fr/cgl-nsi.

3 Statistiques

Cette année, 367 candidats étaient inscrits et 351 étaient présents le jour de l'épreuve. Sur ces candidats inscrits, 325 l'étaient en France et 26 dans des lycées français à l'étranger.

Avec 49 personnes de genre féminin inscrites contre 318 de genre masculin, la proportion du genre féminin reste encore faible (13,9 %) mais elle est en progrès par rapport à la session précédente. Le jury regrette que le nombre de candidates ne soit pas plus élevé, même si cette proportion est comparable à celle des élèves du genre féminin sur l'ensemble des élèves inscrits en NSI en terminale.

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

4 Bilan de la session 2025

Le sujet comporte deux parties indépendantes. La première partie porte sur le problème de la satisfiabilité de formules logiques propositionnelles, problème fondamental en algorithmique notamment pour l'étude des classes de complexité. Le travail sur les booléens dans le programme de la spécialité NSI permet tout à fait d'aborder, au-delà de la seule manipulation des booléens dans du code Python ou dans des circuits logiques, le début de la logique propositionnelle.

La deuxième partie porte sur des méthodes de tri avec certaines contraintes, et sa résolution repose sur la maîtrise de structures de données usuelles telles que les piles et les files, et la maîtrise de techniques algorithmiques telles que la programmation dynamique.

L'ensemble du sujet accorde une part importante à la maitrise de la programmation en Python.

Chaque partie est assez longue, permettant aux candidates et aux candidats de s'investir dans la résolution de chaque partie grâce aux outils et connaissances du programme. Les meilleures copies ont pu traiter le sujet presque en totalité.

5 Palmarès

Le jury a retenu 18 copies, conformément aux règles du Concours général des lycées, les estimant d'une qualité remarquable.

Il note que parmi les autres copies, un très grand nombre de copies étaient d'un niveau très bon ou excellent, ce qui témoigne d'un très bon niveau de préparation des candidates et des candidats.

Le jury félicite donc 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.

6 Conseils généraux aux futurs candidates et candidats

6.1 Présentation des copies

Le soin apporté à l'écriture, qui doit être lisible, et la présentation, en limitant notamment la quantité de ratures ou de rajouts entre les lignes, est essentiel pour faciliter le travail de lecture des copies et d'évaluation par le jury. Le jury conseille d'exploiter un brouillon pour aider à la réflexion et préparer les éléments de réponse avant leur rédaction.

En particulier, 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. De surcroît, le jury apprécie les copies présentant une progression nette dans les différentes parties, démontrant que l'idée d'ensemble de la partie est comprise.

Le jury considère qu'il est souvent plus facile d'aborder la rédaction des questions dans l'ordre de l'énoncé, quitte à parfois passer une question en laissant un blanc sur la copie, afin de se laisser la possibilité de compléter plus tard.

6.2 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.

7 Remarques détaillées par question

Question 1

Le but de cette question et des suivantes est de s'assurer que les candidats ont compris le contexte du sujet et les définitions qui précèdent. Les notions mobilisées appelant des notions du programme de la classe de première, le but est d'assurer une entrée progressive dans le sujet.

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

Pour la formule C, deux réponses étaient possibles. Il suffisait d'en donner une des deux.

Question 2

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

Question 3

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

Question 4

Cette question est abordée dans la quasi-totalité des copies, mais avec des rédactions plus ou moins rigoureuses. L'énoncé demandant explicitement une justification, il ne suffisait pas d'indiquer sans autre précision que les égalités découlaient des tables de vérité. Sélectionner les lignes pertinentes de chaque table est, par exemple, une approche mieux argumentée.

Question 5a

Cette question est traitée correctement dans la quasi-totalité des copies. Elle vise uniquement à s'assurer de la bonne compréhension de la notion de forme normale conjonctive et de la structure Python utilisée.

Question 5b

Les réponses à cette question sont partagées, entre certains candidats qui comprennent immédiatement les conventions de représentation des variables en Python, et d'autres qui sont assez largement hors sujet.

Question 6

Cette première question de programmation est traitée par la quasi-totalité des candidats. Elle présente plusieurs difficultés, qui permettent de distinguer les copies entre elles. Il faut en particulier faire attention aux points suivants :

- quand une clause contient le littéral à simplifier, il faut retirer la clause et ne pas insérer une clause vide dans le résultat ;
- il faut traiter correctement la présence du littéral opposé dans la clause, en le retirant. Dans le cas d'une clause réduite à ce littéral opposé, le résultat doit être la clause vide.

La plupart des candidats ont bien respecté la spécification de l'énoncé qui demandait de créer une copie de la formule. Certains candidats ont cependant réutilisé les clauses de la formule d'origine, sans les recopier, quand elles ne contenaient ni le littéral ni son opposé. Ce cas n'a pas été sanctionné car la correction de la fonction n'en dépendait pas.

Même s'il est avantageux de tester l'appartenance du littéral en premier afin d'éviter de parcourir toute la clause lorsque celui est présent, le jury n'a pas évalué ce point lors de la correction.

Question 7

Le sujet ne spécifiait pas le comportement à adopter lorsque l'on aboutissait à une contradiction. Les candidats étaient donc libres d'adopter celui qu'ils souhaitaient. Dans une telle situation, une bonne habitude de programmation consiste à préciser (rapidement) le choix qui est fait.

Dans cette question, on trouve fréquemment une erreur de programmation majeure. Des candidats itèrent sur les clauses de la formule et modifient la formule elle-même durant cette itération :

```
for clause in formule:
    if len(clause) == 1:
        formule = simplifie(formule, clause[0]) # Modification non autorisée
        litteraux.append(clause[0])
```

Il faut également faire attention au fait que des clauses qui n'étaient pas unitaires initialement peuvent le devenir au cours de la simplification.

Pour ces deux raisons, une simple boucle `for` itérant sur les clauses de la formule ne peut pas convenir.

De nombreuses copies présentent cependant des codes corrects, et parfois assez différents d'une copie à l'autre (version itérative qui recense les clauses unitaires et les simplifie toutes puis recommence, version itérative qui cherche une clause unitaire à simplifier puis recommence, version récursive).

Attention toutefois lorsqu'on recense toutes les clauses unitaires avant de les simplifier : comme précisé dans l'énoncé, il faut éviter les doublons dans la solution. Le résultat sur la formule `[[3],[-3,4,5],[3],[6,7]]` doit être `([[4,5],[6,7]], [3])` et non pas `([[4,5],[6,7]], [3,3])`.

Question 8

Pour cette question, les erreurs les plus fréquemment rencontrées sont :

- des difficultés à trouver la valeur du littéral. Des candidats ont confondu le littéral avec la variable, ce qui donne lieu à l'utilisation incorrecte d'indices négatifs dans le tableau des valeurs ;
- la gestion des cas `None` est source d'erreurs. De nombreux candidats ont cru à tort que sa présence dans un terme rendait possible de renvoyer immédiatement une valeur pour une disjonction ou une conjonction. À contrario, de nombreux candidats ont su à bon escient interrompre des boucles dès qu'il était possible de conclure (un vrai dans une disjonction ou un faux dans la conjonction) ;

— des candidats ont itéré sur l'ensemble de toutes les variables, aboutissant à des fonctions dont la complexité est plus grande que linéaire en la taille de la formule ;

Pour une telle fonction, il peut être utile de prendre l'initiative de la décomposer en fonctions intermédiaires. Cela donne un code plus clair, d'autant plus lorsque les candidats documentent d'une phrase rapide les fonctions auxiliaires.

Question 9

Cette question permet de s'assurer que les candidats ont bien compris l'algorithme énoncé. Le jury insiste sur le fait que ce type de question semble fastidieux aux yeux de certains candidats, mais que traiter cette question avec soin permet souvent de comprendre de nombreuses subtilités importantes pour la suite.

Question 10a

Il s'agit d'une question de cours, destinée d'une part à rassurer les candidats, en entracte entre deux parties du sujet plus compliquées, et d'autre part à valoriser les connaissances du programme de la spécialité.

Ici, une justification rapide est appréciée, surtout pour une épreuve de haut niveau telle que le concours général.

Question 10b

Cette question ne pose pas de difficulté particulière. Comme en question 8, il ne faut pas confondre le littéral et la variable.

Question 11

Les réponses à cette question sont très souvent partielles. De nombreux candidats traitent à peu près correctement le cas de la dernière variable, soit en modifiant sa valeur soit en la retirant si sa valeur n'est plus modifiable, mais ne pensent pas à dépiler les variables qui ne sont pas modifiable dans `self.Ordre`.

Question 12a

Du fait d'une erreur de balisage dans la source du sujet, la référence à la propriété 2 n'était pas correcte dans l'énoncé. La plupart des candidats ont rectifié cette erreur spontanément.

Question 12b

Cette question nécessite d'avoir bien compris la méthode proposée par le sujet. C'est le cas de la plupart des candidats, mais on note cependant plusieurs réponses hors sujet.

Question 12c

Remarque similaire à la question 12b.

Question 12d

Remarque similaire à la question 12b.

Question 13

Question de programmation qui nécessite un peu d'autonomie.

Question 14a

Cette question a été très souvent bien traitée par les candidats. On attend un argument qui évoque le temps de calcul de l'appel à `valeur_formule`.

Question 14b

Des candidats ont proposé un test qui n'était pas en temps constant.

Question 16

Cette question très ouverte et difficile n'a pratiquement pas été traitée.

Question 17

Question 15a

On note deux erreurs fréquentes :

— la confusion entre variable et littéral, il faut faire attention aux signes des entiers manipulés,
— une mauvaise gestion des doublons, selon la manière dont on choisit d'initialiser.

Cette question est peu traitée et très rarement réussie. Compte tenu des attendus du programme de la spécialité, l'énoncé aurait dû rappeler l'usage de `GROUP BY` avec les fonctions d'agrégation SQL.

Sur les copies où cette question est abordée, même si la requête ne renvoie pas le résultat attendu, le jury note que la syntaxe générale de SQL est néanmoins acquise.

Question 18

Cette question est similaire à la précédente.

Question 19

Cette question est très compliquée à traiter. Le jury a noté quelques tentatives, rares mais pertinentes.

Question 20

Le jury a accepté une formule donnée en forme normale conjonctive ou bien sa représentation en Python.

On note parfois quelques confusions sur les indices (inversion ligne et colonne, mauvaise numérotation).

Question 21

La principale, petite, difficulté consiste à bien placer les retours à la ligne dans l'affichage.

Question 22

Le jury a accepté une forme normale conjonctive ou bien tout code Python qui créait la bonne représentation.

Question 23

On note ici encore plusieurs erreurs sur la manipulation des indices ou le placement des retours à la ligne.

Question 27

Cette question a été traitée correctement par la quasi-totalité des candidats.

Question 28

Cette question a été traitée correctement par la quasi-totalité des candidats.

Question 29a

La bonne réponse est donnée par la majorité des candidats.

Question 29b

Les réponses à cette question ont été très différentes, entre les candidats qui voient bien l'unicité de la séquence de mouvements, ceux qui ne la voient pas, et ceux qui interprètent incorrectement la question.

Une justification rapide est appréciée.

Question 30

De nombreuses réponses considèrent une séquence particulière d'actions, puis constatent qu'une fois cette séquence effectuée, il n'est plus possible de terminer le tri. Cette démarche n'est pas correcte, car elle ne démontre pas qu'en démarrant autrement il n'aurait pas été possible de trier.

Les meilleures copies parviennent à donner des réponses bien construites.

Question 31

L'algorithme a été compris par une très large majorité des candidats. On note cependant plusieurs erreurs fréquentes, dues essentiellement à des tests erronés dans les boucles `while` :

- dépiler des wagons alors que la voie d'entrée est vide ;
- une fois que la voie d'entrée est vide, oublier de vider la voie de sortie.

Question 32a

Cette question est traitée correctement par la majorité des candidats.

Question 32b

Pour cette question, une justification est fortement appréciée.

Question 32c

Aucune réponse totalement correcte n'a été donnée pour cette question plus difficile.

Une très grande partie des candidats propose des trains de longueurs inférieures où égale à 6 wagons, ce qui a étonné le jury étant donné que le sujet admet que de tels trains sont tous triables.

Parmi les candidats qui proposent des trains à 7 wagons, la plupart ne justifie pas que ces trains ne sont pas triables (et, hélas, ils étaient presque tous triables).

Très peu de candidats se sont lancés dans une preuve rigoureuse. Le jury a valorisé les copies où une telle tentative était présente. Quelques très rares candidats ont rédigé une preuve qui leur a en fait permis de constater que la liste qu'ils avaient proposée était triable. Si cela ne constituait pas une preuve parfaite, la réponse a cependant été valorisée : à l'exception d'une petite inversion sur l'ordre des wagons, tous les ingrédients d'une excellente réponse étaient présents.

Une réponse par un argument de dénombrement était également possible, mais nécessite des connaissances en mathématiques qui dépassent le programme de l'enseignement de spécialité NSI.

Question 33

Cette question de cours est destinée à valoriser les connaissances du cours de spécialité. De nombreux candidats proposent pour cette question une structure de graphe, ce qui n'est pas la réponse attendue.

Un graphe orienté serait une structure pertinente pour représenter tout un réseau de triage, où les sommets seraient les voies et les arcs seraient les raccordements entre ces voies (voir [Tarjan, Sorting

Using Networks of Queues and Stacks]). Cela ne correspond cependant pas à la question posée, à savoir la représentation d'une voie seule.

Question 34

Cette question est en général bien traitée, malgré une erreur assez fréquente : de nombreux candidats utilisent le fait que la présence de deux wagons dans le mauvais ordre sur une même voie entraîne une contradiction. C'est une conséquence du résultat de la question 36, mais il n'est à ce stade pas encore démontré.

Question 35

Par rapport à la question 34, il est important de bien gérer la généralisation du résultat :

— ce n'est plus le train complet qui est strictement décroissant, il faut expliquer que si un train est triable alors tout sous-train l'est également ;
— alors on peut se servir de la question précédente pour un sous-train de taille $m+1$. La généralisation du résultat à 10 wagons de la question précédente à $m+1$ wagons a juste besoin d'être rapidement mentionnée. Elle est immédiate et n'appelle pas de nouvelle preuve supplémentaire très détaillée.

Question 36

L'algorithme est en général bien compris par les candidats qui traitent cette question et sa description est très souvent satisfaisante.

En revanche, très peu de copies traitent la première partie de la question et omettent donc d'expliquer qu'il est correct de travailler en deux phases : d'abord remplir les voies, puis les vider.

Question 37

Lorsqu'elle est traitée, cette question est très souvent correcte, les candidats parvenant avec aisance à manipuler les structures données par l'énoncé.

Question 38

On attend la disjonction de cas qui explique comment fabriquer un train à partir des sous-cas. La construction des trains dans les différents cas doit être claire.

Question 39

On attend une construction explicite d'un train à partir d'un autre.

Question 40

L'énoncé guidait, avec les questions précédentes, une solution en programmation dynamique. On note, sur le thème de la programmation dynamique, une vraie progression par rapport aux questions similaires de la session 2024.

Il était également possible de traiter cette question sans programmation dynamique, en utilisant l'algorithme de tri des wagons qui crée les voies de triage au fur et à mesure qu'on en a besoin, puis compte ces voies.

Question 41

On attend d'une part une complexité cohérente avec le code écrit en question précédente. Une complexité annoncée sans aucun lien avec du code écrit ne peut pas être considérée comme juste.

8 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 1

Pour A, on peut choisir :

— $x_1 = 0, x_2 = 0$,
— $x_1 = 0, x_2 = 1$,
— $x_1 = 1, x_2 = 1$.

Pour B, on peut uniquement choisir $x_1 = 0, x_2 = 1$.

Pour C, il y a 2 solutions :

— $x_1 = 1, x_2 = 0, x_3 = 0$,
— $x_1 = 0, x_2 = 1, x_3 = 0$.

Question 2

On peut par exemple prendre $x \wedge \neg x$.

Question 3

x_1	x_2	$x_1 \vee x_2$
0	0	0
0	1	1
1	0	1
1	1	1

Question 4

Toutes les égalités peuvent être vérifiées par inspection des tables.

Ainsi, on a $\underline{0} \vee 0 = \underline{0}$ et $\underline{1} \vee 0 = \underline{1}$, ce qui implique que $\underline{x} \vee 0 = \underline{x}$.

L'égalité $x \vee 1 = 1$ se déduit des lignes 2 et 4 de la table donnée en question précédente.

L'égalité $x \wedge 0 = 0$ se déduit des lignes 1 et 3 de la table donnée par le sujet.

L'égalité $x \wedge 1 = x$ se déduit des lignes 2 et 4 de la table donnée par le sujet.

Question 5a

On obtient la formule : $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_5 \vee x_3) \wedge (\neg x_3) \wedge (x_5 \vee \neg x_2)$.

Question 5b

Si on autorisait le nombre 0 pour numéroter les variables, on ne pourrait pas faire la différence entre x_0 et $\neg x_0$.

Question 6

```
def simplifie(F, lit):
    """renvoie la formule ``F`` simplifiée lorsqu'on suppose que le littéral
       ``lit`` devient vrai :
      - supprime les clauses qui contiennent ``lit`` (elle sont vraies)
```

```
      - supprime les littéraux ``-lit`` dans les autres clauses (ce littéral
        est faux)
    """
    simplified_F = []
    for cl in F:
        if lit in cl:
            continue
        simplified_F.append([x for x in cl if x != -lit])
    return simplified_F
```

Question 7

```
def simplifie_clauses_unitaires(F):
    """simplifie la formule avec la propagation des clauses unitaires
    La fonction renvoie une paire contenant la formule simplifiée et la liste
    des littéraux simplifiés.
    """
    sol = []
    while True:
        lit = 0     # litéral de la première clause unitaire (0: valeur sentinelle)
        for cl in F:
            if len(cl) == 1:
                lit = cl[0]
                sol.append(lit)
                break
        if lit == 0:
            break
        F = simplifie(F, lit)
    return F, sol
```

Question 8

On introduit deux méthodes auxiliaires pour simplifier l'écriture du code, dont la spécification est donnée en documentation pour chacune.

```
    def valeur_lit(self, lit):
        """renvoie la valeur True / False / None d'un littéral dans la solution donnée
        None correspond aux littéraux indéterminés, càd qui ne font pas partie de
        la solution"""
        v = self.Valeur[abs(lit)]
        if v is None:
            return None
        elif lit > 0:
            return v
        else:
            return not v

    def valeur_clause(self, cl):
        """renvoie la valeur True / False / None d'une clause pour la solution donnée
        None correspond aux clauses indéterminées, càd qui contiennent au moins un
        littéral indéterminé et aucun littéral vrai"""
        r = False
        for lit in cl:
            v = self.valeur_lit(lit)
            if v is None:
                r = None
```

```
            elif v is True:
                return True
        return r

    def valeur_formule(self):
        """renvoie la valeur True / False / None d'une formule pour la solution
        donnée
        None correspond aux formules indéterminées, càd qui contiennent au moins
        une clause indéterminée et aucune clause fausse"""
        r = True
        for cl in self.Cl:
            v = self.valeur_clause(cl)
            if v is None:
                r = None
            elif v is False:
                return False
        return r
```

Question 9

On obtient :

- `F.Valeur = [_, True, None, None]` ;
- `F.Ordre = [1]` ;
- `F.Reste = [_, True, False, False]`.

Question 10a

L'attribut `Ordre` stocke des variables en conservant l'ordre d'ajout et les restitue dans l'ordre contraire de leur ajout (la dernière variable ajoutée est la première retirée). Ceci correspond à une structure de pile.

Question 10b

```
    def ajoute_lit(self, lit):
        x = abs(lit)
        self.Ordre.append(x)
        self.Valeur[x] = lit > 0
        self.Reste[x] = True       # il restera encore une valeur à tester (False)
```

Question 11

```
    def backtrack(self):
        """revient à la dernière décision prise que l'on peut modifier,
        __en modifiant ``Valeur`` et ``Ordre``__
        Si on ne peut revenir sur aucune décision, ``Ordre`` deviendra vide"""
        while self.Ordre:
            x = self.Ordre.pop()
            if self.Reste[x]:
                # si on peut modifier la valeur : on le fait
                self.Valeur[x] = not self.Valeur[x]
                # et on note cette décision comme non modifiable
                self.Ordre.append(x)
                self.Reste[x] = False
                return True
            else:
                # sinon, il faut revenir plus en arrière dans la solution
                self.Valeur[x] = None
```

```
                # il n'est pas nécessaire de mofifier Reste[x]
        return False
```

Question 12a

Une formule en FNC est fausse exactement lorsqu'une des clauses ne contient que des littéraux faux. Il est donc toujours possible, pour une formule non fausse, de trouver un littéral non faux dans chaque clause. On peut mettre ce littéral en tête de clause pour satisfaire la propriété « aucune clause de F ne commence par un littéral faux ».

Réciproquement, dès que la formule devient fausse, une de ses clauses (au moins) ne contient que des littéraux faux et on ne peut plus rétablir la propriété.

Question 12b

Avant d'introduire la variable `x` dans la solution partielle, la formule satisfaisait la propriété « aucune clause de F ne commence par un littéral faux ». Pour toutes les clauses commençant par un littéral différent de `x` et `-x`, la valeur introduite pour `x` ne modifie pas cette propriété. Si l'on souhaite rétablir la propriété, il suffit donc de regarder les clauses commençant par `x` ou `-x`, c'est-à-dire les clauses listées dans `Surveile[x]` et `Surveille[-x]`.

Question 12c

Si la variable `x` devient vraie, alors toutes les clauses commençant par `x` commencent par un littéral vrai, et donc par un littéral non faux. Seules les clauses commençant par `-x` rendent la propriété « aucune clause de F ne commence par un littéral faux » fausse.

Pour rétablir la propriété, il suffit de parcourir ces clauses à la recherche d'un littéral non faux. S'il y a en un, on peut l'échanger avec `-x` pour rétablir la propriété. S'il n'y en a pas, c'est qu'on est sur une clause dont tous les littéraux sont faux. La formule est fausse et la méthode `non_fausse` peut renvoyer `False`.

Question 12d

Le raisonnement est similaire lorsque la variable `x` devient fausse. Il suffit de remplacer `x` par `-x` dans les explications précédentes.

Question 13

```
    def non_fausse(self):
        """renvoie la valeur True / False d'une formule pour la solution
        partielle courante
        Attention : True correspond aux formules vraies __ou indéterminées__
        La fonction rétablit également l'invariant que le premier littéral de chaque
        clause est non faux.
        """
        if not self.Ordre:
            return None

        x = self.Ordre[-1]
        lit = x if self.Valeur[x] else -x

        # Lorsque lit devient vrai, ce sont les clauses de Active[-lit] qu'il faut
        # mettre à jour. Les clauses de Active[lit] deviennent toutes vraies, et
        # l'invariant reste vrai

        # on traite les clauses de Active[-lit] en vidant Active[-lit] par la fin, pour
        # éviter les incohérences pendant la boucle
```

```
        for idx_cl in reversed(self.Active[-lit]):
            cl = self.Cl[idx_cl]

            # on cherche un autre litéral non faux dans la clause
            for j in range(1, len(cl)):
                new_lit = cl[j]
                new_v = self.valeur_lit(new_lit)
                if new_v is not False:
                    # on échange la première clause (-lit) avec la clause n°j (new_lit)
                    cl[j] = -lit
                    cl[0] = new_lit
                    # cette clause est maintenant surveillée par new_lit
                    self.Active[new_lit].append(idx_cl)
                    # et elle n'est plus surveillée par -lit
                    assert self.Active[-lit][-1] == idx_cl
                    self.Active[-lit].pop()
                    break
            else:   # on a fini la boucle sans trouver de littéral non faux
                # la clause est donc fausse, et la formule aussi
                # (il faudra bactracker...)
                return False
        # si on est passé par toutes les clauses, c'est que la formule est vraie ou
        # indéterminée
        return True
```

Question 14a

On essaie justement de supprimer les appels à la méthode `valeur_formule` initiale pour améliorer la complexité. Si on appelle `valeur_formule` à chaque étape de la boucle, nous ne gagnerons rien.

Question 14b

Il suffit de regarder la taille du tableau `Ordre` : lorsqu'il contient toutes les variables, et que la formule est non fausse, elle est forcément vraie.

On peut donc avoir un test `if len(F.Ordre) == F.nb_variables`.

Question 15a

```
        # initialisation de Active
        self.Active = {}
        for x in range(1, self.nb_variables+1):
            self.Active[x] = []
            self.Active[-x] = []

        for i in range(len(F)):
            self.Active[F[i][0]].append(i)
```

Question 15b

Pour que `lit` soit une clause unitaire, tous les autres littéraux de la clause doivent être faux. Il faut donc que cette clause soit dans `Surveille[lit]`.

La méthode `clause_unitaire` va donc faire une boucle sur les variables actives (`for x in Active`), puis un parcours des clauses de `Surveille[x]` et `Surveille[-x]`. Pour chacune de ses clauses, il va falloir regarder les autres littéraux de la clause : s'ils sont tous faux (fonction `valeur_lit`), on a trouvé une

clause unitaire et on peut la renvoyer. Si on trouve un autre littéral non faux, la clause n'est pas unitaire et on passe à la clause unitaire.

Lorsqu'on trouve une clause unitaire, on peut anticiper la suite et la supprimer directement de la liste `Active`, par exemple en l'échangeant avec la dernière variable avant de faire un `Active.pop()`. Ceci évitera un parcours de `Active` supplémentaire.

```
    def clause_unitaire(self):
        """renvoie le littéral correspondant à une clause unitaire ; ou 0
        lorsqu'il n'y en a pas
        Lorsqu'une clause unitaire est trouvée, la variable correspondante est
        supprimée de la liste des variables actives (Active)"""
        # on regarde toutes les variables actives
        for ix, x in enumerate(self.Active):
            # et les 2 littéraux ±x correspondant
            for lit in [-x, +x]:
                # on cherche une clause commençant par lit, qui serait unitaire
                for idx_cl in self.Surveille[lit]:
                    cl = self.Cl[idx_cl]
                    assert lit == cl[0]
                    v = self.valeur_lit(lit)
                    assert v is None
                    # on doit vérifier que tous les autres literaux sont faux
                    for j in range(1, len(cl)):
                        other_lit = cl[j]
                        v = self.valeur_lit(other_lit)
                        if v is not False:
                            break
                    else:   # si tous les littéraux étaient faux, la clause est une clause unitaire
                        # on renvoie le premier littéral (lit)
                        # avant ça, on supprime (par anticipation) la variable
                        # correspondante de la liste des variables actives
                        self.Active[ix] = self.Active[-1]
                        self.Active[-1] = ix
                        self.Active.pop()
                        return lit
        return 0
```

Question 15c

Il faut modifier la fonction `backtrack` : lorsqu'on supprime une variable `x` de la solution, cette variable peut devenir active si `Surveille[x]` ou `Surveille[-x]` est non vide.

```
    def backtrack(self):
        """revient à la dernière décision prise que l'on peut modifier,
        __en modifiant ``Valeur`` et ``Ordre``__
        Si on ne peut revenir sur aucune décision, ``Ordre`` deviendra vide"""
        while self.Ordre:
            x = self.Ordre.pop()
            if self.Reste[x]:
                # si on peut modifier la valeur : on le fait
                self.Valeur[x] = not self.Valeur[x]
                # et on note cette décision comme non modifiable
                self.Ordre.append(x)
                self.Reste[x] = False
                return True
            else:
```

```
                # sinon, il faut revenir plus en arrière dans la solution
                self.Valeur[x] = None
                # la variable x devient active si elle apparait dans le premier
                # littéral d'une clause
                if self.Surveille[x] or self.Surveille[-x]:
                    self.Active.append(x)
        else:
            return False
```

Question 15d

Il faut modifier la fonction `non_fausse` : lorsqu'on échange le premier littéral (devenu faux) d'une clause avec un autre de la même clause, la variable correspondante `x` peut devenir active. Pour savoir si la variable était déjà active, il suffit de regarder `Surveille[x]` ou `Surveille[-x]` est non vide.

```
    def non_fausse(self):
        """renvoie la valeur True / False d'une formule pour la solution
        partielle courante
        Attention : True correspond aux formules vraies __ou indéterminées__
        La fonction rétablit également l'invariant que le premier littéral de chaque
        clause est non faux.
        """
        if not self.Ordre:
            return None

        x = self.Ordre[-1]
        lit = x if self.Valeur[x] else -x

        # Lorsque lit devient vrai, ce sont les clauses de Surveille[-lit] qu'il faut
        # mettre à jour. Les clauses de Surveille[lit] deviennent toutes vraies, et
        # l'invariant reste vrai

        # on traite les clauses de Surveille[-lit] en vidant Surveille[-lit] par la fin, pour
        # éviter les incohérences pendant la boucle
        for idx_cl in reversed(self.Surveille[-lit]):
            cl = self.Cl[idx_cl]

            # on cherche un autre litéral non faux dans la clause
            for j in range(1, len(cl)):
                new_lit = cl[j]
                new_x = abs(new_lit)
                new_v = self.valeur_lit(new_lit)
                if new_v is not False:
                    # on échange la première clause (-lit) avec la clause n°j (new_lit)
                    cl[j] = -lit
                    cl[0] = new_lit

                    # la nouvelle variable en tête de la clause devient active, si
                    # elle ne l'était pas
                    if new_v is None and not self.Surveille[new_x] and not self.Surveille[-new_x]:
                        self.Active.append(new_x)

                    # cette clause est maintenant surveillée par new_lit
                    self.Surveille[new_lit].append(idx_cl)
                    # et elle n'est plus surveillée par -lit
                    assert self.Surveille[-lit][-1] == idx_cl
```

```
                self.Surveille[-lit].pop()
                break
        else:   # on a fini la boucle sans trouver de littéral non faux
            # la clause est donc fausse, et la formule aussi
            # (il faudra bactracker...)
            return False
    # si on est passé par toutes les clauses, c'est que la formule est vraie ou
    # indéterminée
    return True
```

Question 16

Les listes `F.Surveille` peuvent permettre de surveiller les 2 premiers littéraux de chaque clause.

Initialement, chaque `F.Surveille[x]` (resp. `F.Surveille[-x]`) contient les indices des clauses avec `x` (resp. `-x`) en première ou seconde position. Pour éviter des problèmes avec les clauses qui ne contiennent qu'un seul littéral, on peut commencer par les simplifier en utilisant la question 7 (simplification des clauses unitaires). Cette étape de pré-traitement garantit que toutes les clauses contiennent au moins 2 littéraux.

La liste des variables actives contient comme précédemment les variables `x` t.q. `F.Surveille[x]` ou `F.Surveille[-x]` est non vide.

La méthode `non_fausse` fonctionne de la même manière : lorsque `x` prend la valeur "vraie", on parcourt les clauses de `F.Surveille[-x]` pour chercher un nouveau littéral non faux à mettre en position 0 ou 1, pour remplacer `-x`.

À la différence de la version précédente, si cela n'est pas possible, c'est que la clause est unitaire : un de ces 2 premiers littéraux est non faux (celui différent de `-x`) et tous les autres sont faux.

On peut donc détecter et traiter les clauses unitaires directement dans la méthode `non_fausse`. On conserve une liste de littéraux à traiter : cette liste est initialisée avec le dernier littéral ajouté à la solution, calculé à partir de `F.Ordre` et `F.Valeur`

Lorsqu'on recherche un nouveau littéral à mettre en position 0 ou 1 : - soit on en trouve un, et on continue à traiter les littéraux de la liste,

— soit on n'en trouve pas, et l'autre littéral (différent de celui que l'on est en train de traiter) est non faux : la clause est une clause unitaire et on peut forcer la valeur de ce dernier littéral non faux de la clause en mettant `F.Valeur` et `F.Ordre` à jours et en l'ajoutant dans la liste des littéraux à traiter.

— soit on n'en trouve pas, et l'autre littéral est faux (car on l'a traité plus tôt) : la clause devient donc fausse et on ne peut pas rétablir l'invariant. La méthode `non_fausse` renvoie `False`.

Le temps de calcul la méthode `non_fausse` devient plus élevé mais elle fait automatiquement les simplification des clauses unitaire, *sans parcourir toutes les clauses de la formule.*

Question 17

```
select id_formule, count(distinct id_clause) as nb_clauses
  from litteral group by id_formule
```

Question 18

```
select id_formule, count(distinct abs(num_litteral)) as nb_variables
  from litteral group by id_formule
```

Question 19

```sql
select nb_clauses/nb_variables as quotient, count(f.id_formule) as nb_formules
from
  (select id_formule, count(distinct id_clause) as nb_clauses
      from litteral group by id_formule) nv
  join (select id_formule, count(distinct abs(num_litteral)) as nb_variables
          from litteral group by id_formule) nf
    on nv.id_formule = nf.id_formule
  join formule f on nv.id_formule = f.id_formule
where f.est_sat
group by quotient
order by quotient;
```

Question 20

La contrainte « la ligne n°1 contient au moins une fois le nombre 2 » contient une unique clause avec 4 littéraux : (case_1_1_contient_2 $\vee$ case_2_1_contient_2 $\vee$ case_3_1_contient_2 $\vee$ case_4_1_contient_2).

Question 21

```python
for n in range(1, 5):
    for j in range(1, 5):
        for i in range(1, 5):
            print(f"case_{i}_{j}_contient_{n}", end=" ")
        print()
```

Question 22

La contrainte « la ligne n°1 contient au plus une fois le nombre 2 » contient 6 clauses avec 2 littéraux chacune :

$$
\begin{aligned}
&(\neg\texttt{case\_1\_1\_contient\_2} \vee \neg\texttt{case\_2\_1\_contient\_2})\\
\wedge&(\neg\texttt{case\_1\_1\_contient\_2} \vee \neg\texttt{case\_3\_1\_contient\_2})\\
\wedge&(\neg\texttt{case\_1\_1\_contient\_2} \vee \neg\texttt{case\_4\_1\_contient\_2})\\
\wedge&(\neg\texttt{case\_2\_1\_contient\_2} \vee \neg\texttt{case\_3\_1\_contient\_2})\\
\wedge&(\neg\texttt{case\_2\_1\_contient\_2} \vee \neg\texttt{case\_4\_1\_contient\_2})\\
\wedge&(\neg\texttt{case\_3\_1\_contient\_2} \vee \neg\texttt{case\_4\_1\_contient\_2})
\end{aligned}
$$

Question 23

En Python :

```python
for n in range(1, 5):
    for i in range(1, 5):
        for j in range(1, 5):
            for k in range(i+1, 5):
                print(f"-case_{i}_{j}_contient_{n} -case_{i}_{k}_contient_{n}")
```

Question 24a

Pour une case (i, j), la clause est la suivante :

$$\texttt{case\_i\_j\_contient\_1} \vee \texttt{case\_i\_j\_contient\_2} \vee \texttt{case\_i\_j\_contient\_3} \vee \texttt{case\_i\_j\_contient\_4}$$

Question 24b

On considère la case (i, j). Chaque case contient au plus un entier lorsque la FNC suivante est satisfaite :

$$\begin{aligned}
&(\neg \texttt{case_i_j_contient_1} \vee \neg \texttt{case_i_j_contient_2}) \\
\wedge\ &(\neg \texttt{case_i_j_contient_1} \vee \neg \texttt{case_i_j_contient_3}) \\
\wedge\ &(\neg \texttt{case_i_j_contient_1} \vee \neg \texttt{case_i_j_contient_4}) \\
\wedge\ &(\neg \texttt{case_i_j_contient_2} \vee \neg \texttt{case_i_j_contient_3}) \\
\wedge\ &(\neg \texttt{case_i_j_contient_2} \vee \neg \texttt{case_i_j_contient_4}) \\
\wedge\ &(\neg \texttt{case_i_j_contient_3} \vee \neg \texttt{case_i_j_contient_4})
\end{aligned}$$

Question 24c

En Python, cela donne :

```python
for i in range(1, 5):
    for j in range(1, 5):
        for num in range(1, 5):
            print(f"case_{i}_{j}_contient_{num} ", end="")
        print()
        for num1 in range(1, 5):
            for num2 in range(num1 + 1, 5):
              print(f"-case_{i}_{j}_contient_{num1} -case_{i}_{j}_contient_{num2}")
```

Question 25

Supposons que les contraintes de E_2 et I_1 sont vérifiées. Nous pouvons montrer que celles de I_2 et deE_1 le sont également.

- Pour montrer I_2, il faut montrer que chaque case contient au plus un entier entre 1 et 4. Supposons par contradiction qu'une case contient 2 (ou plus) entiers. Comme toutes les cases de la ligne contiennent au moins un entier (par I_1), la ligne contiendrait 5 entiers (ou plus). Par le lemme des tiroirs, cette ligne contiendrait au moins un entier en double. C'est en contradiction avec E_2.
- Pour montrer E_1, il faut montrer que chaque ligne [colonne / petit carré] contient au moins une occurrence de chaque entier entre 1 et 4. Supposons que la première ligne ne contient pas l'entier 2. Comme toutes les cases de cette ligne contiennent au moins un entier entre 1 et 4 (par I_1), on a forcément un entier répété. (C'est encore le lemme des tiroirs.) C'est en contradiction avec E_2.

Question 26

Les clauses ajoutées peuvent faire apparaître des clauses unitaires plus rapidement, et provoquer plus de simplifications. On peut imaginer le cas extrême où l'on ajouterait explicitement la solution (autant de clauses unitaires qui sont des conséquences logiques des autres contraintes). La simplification des clauses unitaires donnerait alors directement la solution sans aucun besoin de la chercher.

Question 27

La suite d'actions suivantes convient :

- on met 5 sur la voie de stockage,
- on met 0 sur la voie de stockage,
- on sort 0 de la voie de stockage,
- on met 2 sur la voie de stockage,
- on met 1 sur la voie de stockage,
- on sort 1 de la voie de stockage,
- on sort 2 de la voie de stockage,
- on met 4 sur la voie de stockage,

— on met 3 sur la voie de stockage,
— on sort 3 de la voie de stockage,
— on sort 4 de la voie de stockage.

Question 28

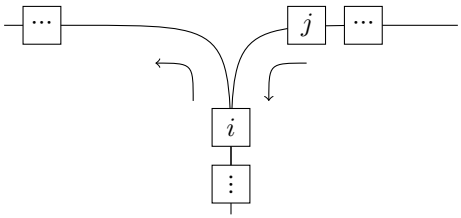
Une voie de stockage permet d'ajouter des wagons, de retirer des wagons, en se souvenant de l'ordre. Elle restitue les wagons dans l'ordre contraire de leur ajout. Elle peut donc se représenter naturellement par une pile.

Question 29a

Chaque wagon effectue exactement deux mouvements : se déplacer de la voie d'entrée vers la voie de stockage et se déplacer de la voie de stockage vers la sortie. Remettre dans l'ordre un train à n wagons se fait en $2n$ mouvements.

Question 29b

Il n'existe qu'une seule suite de mouvements pour trier un train. En effet, on considère deux séquences de mouvements σ_1 et σ_2. On les exprime comme une suite d'opérations que l'on note E (empiler, mouvement de la voie d'entrée vers celle de stockage) et D (dépiler, stockage vers sortie). On regarde le premier moment où σ_1 et σ_2 diffèrent, c'est-à-dire qu'il existe des séquences de mouvements u, v et v' (v et v' sont de même longueur) telles que : $\sigma_1 = uEv$ et $\sigma_2 = uDv'$. On regarde l'état des voies après avoir exécuté les mouvements de u :



Le wagon i en voie de stockage existe car σ_2 doit pouvoir exécuter l'action D. Le wagon j en voie d'entrée existe car σ_1 doit pouvoir exécuter l'action E. Si on suppose que σ_2 trie les wagons, alors en exécutant σ_2, i se trouve en sortie devant j donc $i < j$. En revanche, si on suppose que σ_2 trie, alors l'action E de σ_2 place j dans la pile de stockage au-dessus de i, donc j sera placé en sortie avant i, donc $j < i$. Il est donc contradictoire de dire que les deux séquences trient le train d'entrée. La séquence de tri est donc unique.

Question 30

Soit T un train de type 120. On note $i < j < k$ trois wagons qui apparaissent ainsi dans $T = (\ldots, j, \ldots, k, \ldots, i, \ldots)$.

On suppose que T est triable avec la voie de stockage. On a donc une suite de mouvements qui trie T. On regarde ce qui se produit dans cette suite de mouvements.

Le wagon j est empilé en premier dans la voie de stockage, car il arrive en tête avant les wagons k et i. Tant que i est dans la voie d'entrée, j doit rester dans la voie de stockage, sinon j serait placé à gauche de i en sortie, dont le train ne serait pas trié.

De la même façon, k doit rester dans la voie de stockage tant que i est dans la voie d'entrée. Il y a donc un moment où j et k sont tous les deux dans la voie de stockage. Or la voie de stockage est une pile, donc j est en dessous de k car j a été empilé en premier.

En dépilant, k sera donc dépilé avant j. Il sera donc à gauche de j en sortie. C'est contraire au fait que la suite d'opérations trie T.

Question 31

On propose la fonction suivante :

```
def trier_train_stockage():
    # Numéro du wagon suivant attendu en sortie
    suivant_attendu = 0

    while suivant_entree() != -1 or suivant_stockage() != -1:
        while suivant_stockage() != suivant_attendu:
            entree_vers_stockage()
        stockage_vers_sortie()
        suivant_attendu += 1
```

Question 32a

Pour remettre dans l'ordre le train $(1, 2, 0)$:

- on met 1 dans la voie de stockage 1,
- on met 2 dans la voie de stockage 1,
- on passe 2 de la voie de stockage 1 vers la voie de stockage 2,
- on met 0 dans la voie de stockage 1,
- on passe 0 de la voie de stockage 1 vers la voie de stockage 2,
- on sort 0 de la voie de stockage 2,
- on passe 1 de la voie de stockage 1 vers la voie de stockage 2,
- on sort 1 de la voie de stockage 2,
- on sort 2 de la voie de stockage 2.

Question 32b

En général, il y a plusieurs manières de procéder. Par exemple, si le train en entrée est déjà trié, on peut :

- pour chaque wagon, le mettre dans la voie 1, puis la voie 2 puis en sortie,
- ou alors pour chaque couple de wagons, mettre le premier en voie 1, le deuxième en voie 1, passer le deuxième en voie 2, passer le premier en voie 2, sortir le premier, sortir le deuxième.

Ce sont deux suites de mouvements différentes. On peut en trouver encore autant de différentes pour chaque sous-suite de wagons consécutifs déjà triés.

Question 32c

Le train $(1, 3, 2, 4, 6, 5, 0)$ ne peut pas être remis dans l'ordre.

En effet, supposons qu'il existe une séquence d'opérations qui trie ce train et montrons que c'est absurde.

On remarque d'abord que la voie de stockage 2 doit toujours être triée dans l'ordre croissant de la tête vers le fond de la pile, sinon aucune séquence ne permet de sortir une petite valeur située vers le fond de la pile avant une grande valeur située plus haut dans la pile.

On remarque ensuite que la voie de stockage 1 se comporte comme une voie d'entrée de la voie de stockage 2, en se ramenant au problème précédent à une seule voie de stockage. Donc la voie de stockage 1 ne peut pas contenir, de la tête vers le fond, un sous-train de type $(1, 2, 0)$, sinon la voie de stockage 2 ne pourrait pas le trier.

Au moment où 0 arrive en tête de la voie d'entrée, puisque c'est le dernier wagon, tous les autres sont encore dans les voies de stockage. La seule séquence possible consiste alors à déplacer 0 en deux mouvements vers la voie de sortie. On se retrouve alors dans la situation où 0 est en sortie, et les autres wagons sont répartis sur les deux voies de stockage.

Dans le cas où 6 se trouverait à ce moment-là en voie de stockage 2, alors, d'après la première remarque, 6 est seul en voie de stockage 2. Par opérations sur les piles, l'état de la voie de stockage 1 est nécessairement,

de la tête vers le fond : $(5, 4, 2, 3, 1)$. Les trois derniers wagons de cette liste sont de type 120, ce qui contredit la deuxième remarque. Le wagon 6 est donc dans la voie de stockage 1.

Dans le cas où 5 se trouverait en voie de stockage 2, alors, puisque 6 est en voie 1, 5 est le seul en voie de stockage 2. L'état de la voie de stockage 1 est alors $(6, 4, 2, 3, 1)$ qui contient encore le même sous-train de type 120. Le wagon 5 est donc en voie de stockage 1.

De la même façon, 4 sera en voie de stockage 1.

Ainsi, lorsque 0 vient d'arriver en voie de sortie, alors la voie de stockage 1 contient au moins les wagons 4, 5 et 6. D'après leur ordre sur la voie d'entrée, ils sont donc placés en voie 1 dans l'ordre $(5, 6, 4)$ depuis la tête. Il s'agit d'un sous-train de type 120, ce qui est impossible.

Question 33

Une voie de triage permet d'ajouter des wagons, de retirer des wagons, en se souvenant de l'ordre. Elle restitue les wagons dans le même ordre que leur ajout. Elle peut donc se représenter naturellement par une file.

Question 34

On considère le train $T = (10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0)$. Il contient 11 wagons, donc il y a au moins deux wagons qui vont passer par la même file parmi les 10 disponibles. On note i le premier wagon qui passe par cette file et j le deuxième wagon qui passe par cette file. Puisque T est dans l'ordre décroissant, on sait que $i > j$. Or, la voie est une file, donc i est sorti avant j, donc le wagon i est plus à gauche que j dans la voie de sortie. Le train n'est pas trié.

Question 35

L'argument est le même en considérant non plus le train complet T mais un sous-train T' de longueur $d(T)$. T' a plus de wagons qu'il n'existe de voies de triage, donc deux wagons de T' vont passer par la même voie de triage. En notant i le numéro du premier et j le numéro du deuxième, on conclut de la même façon qu'en question 34.

Question 36

On considère un algorithme qui, à partir d'un état S à un moment donné, d'abord défile d'une voie i, puis enfile dans une voie j. Alors à partir de l'état S, il est possible d'enfiler en voie i puis de défiler de la voie j. Dans ce cas on arrive dans le même état, quel que soit l'ordre des opérations. On peut donc toujours faire les opérations *enfiler* avant les opérations *défiler* et aboutir au même résultat.

L'algorithme prend les wagons en entrée un par un. Pour chaque wagon il considère les voies de triage de haut en bas et ajoute le wagon à la première voie de triage rencontrée dont la queue est plus petite que ce train.

Question 37

On propose la fonction suivante :

```
def gare_triage(m : int, entree : VoieTriage, sortie : VoieTriage):
    voies = [VoieTriage() for _ in range(m)]
    # On remplit les voies de triage
    while not entree.est_vide():
        wagon = entree.retire()
        # On range le wagon dans la première voie telle que la voie
        # soit dans l'ordre croissant (en partant de la gauche)
        for i in range(m):
            if voies[i].est_vide() or voies[i].queue() < wagon:
                voies[i].ajoute(wagon)
```

```
                break

    # Puis on les vide vers la sortie
    while True:
        # On cherche la voie qui a le plus petit wagon
        voie_min = -1
        for i in range(0, m):
            if not voies[i].est_vide():
                if voie_min == -1:
                    voie_min = i
                elif voies[voie_min].tete() > voies[i].tete():
                    voie_min = i
        if voie_min != -1:
            sortie.ajoute(voies[voie_min].retire())
        else:
            # Toutes les voies de triage sont vides
            return
```

Question 38

Si $T[k] \geq M$, alors le wagon $T[k]$ peut faire partie d'un sous-train de $T[0{:}k+1]$ où tous les wagons sont plus grands que M.

Pour constituer un tel sous-train décroissant :

- soit on choisit de prendre $T[k]$, alors ce wagon est en dernière position. Comme le sous-train est décroissant, cela veut dire que les autres wagons doivent être plus grands que $T[k]+1$, et former un sous-train décroissant de $T[0{:}k]$. Le plus grand possible est de longueur $D(k, T[k]+1)$, donc le sous-train obtenu est de longueur totale $1 + D(k, T[k]+1)$;
- soit on ne prend pas $T[k]$, dans ce cas, le plus long sous-train est obtenu en cherchant un sous-train de $T[0{:}k]$ où tous les wagons sont plus grands que M. On obtient un sous-train de longueur $D(k, M)$.

On retient entre ces deux cas celui qui donne le plus long sous-train, donc on a bien $D(k+1, M) = \max(D(k, M), 1 + D(k, T[k]+1))$.

Question 39

Lorsque $T[k] < M$, on ne peut pas utiliser $T[k]$ pour trouver un sous-train de $T[0{:}k+1]$ où tous les wagons sont plus grands que M. Donc un sous-train cherché est toujours un sous-train de $T[0{:}k]$. Le plus grand possible est de longueur $D(k, M)$.

Question 40

On propose la fonction suivante :

```
def max_decroissant(T):
    n = len(T)
    D = [[0 for i in range(n+1)] for j in range(n+1)]
    for k in range(n):
        for M in range(n):
            if T[k] >= M:
                D[k+1][M] = max(D[k][M], 1 + D[k][T[k]+1])
            else:
                D[k+1][M] = D[k][M]
    return D[n][0]
```

Question 41

On réalise un nombre d'opérations de l'ordre de n^2.