BANQUE MPI INTER-ENS – SESSION 2024
RAPPORT RELATIF À L'ÉPREUVE PRATIQUE D'ALGORITHMIQUE ET DE PROGRAMMATION DU CONCOURS COMMUN DES ÉCOLES NORMALES SUPÉRIEURES

Écoles partageant cette épreuve :
ENS de Lyon, ENS Paris-Saclay, ENS Rennes, ENS Ulm

Coefficients (en pourcentage du total des points du concours MPI) :
- ENS de Lyon : 18,3 %
- ENS Paris-Saclay : 13,2 %
- ENS Rennes : 13,9 %
- ENS Ulm : 13,3 %

Membres du jury :
Adrien Koutsos, Joseph Lallemand, Simon Mauras, Yann Ramusat, Théo Winterhalter

Contenu de ce document

Dans ce rapport, après avoir rappelé l'organisation de l'épreuve, nous faisons quelques remarques générales sur son déroulement, mêlant des conseils qui reviennent d'années en années, et des observations nouvelles. Enfin, nous revenons plus en détail sur chacun des deux sujets, en insistant sur certains points que nous avons jugé marquants dans les sujets ainsi que les réponses des candidats et candidates.

Le début du rapport, jusqu'à la discussion spécifique aux sujets de cette année, est identique dans les rapports de série MP et MPI.

Organisation de l'épreuve

L'objectif de cette épreuve est d'évaluer la capacité à mettre en œuvre une chaîne complète de résolution d'un problème informatique, à savoir la construction d'algorithmes, le choix de structures de données, leur implémentation, et l'élaboration d'arguments mathématiques pour justifier ces décisions. Le déroulement de l'épreuve est le suivant : un travail sur machine d'une durée de 3 h 30, immédiatement suivi d'une présentation orale pendant 22 minutes.

Juste avant la distribution des sujets, les candidats et candidates disposent d'une période de 10 minutes pour se familiariser avec l'environnement informatique et poser des questions en cas de difficultés d'ordre pratique.

Un sujet contient typiquement une dizaine de questions écrites et une dizaine de questions orales. Il commence généralement par des questions de programmation simples, ayant pour objet la génération des données d'entrée du problème étudié, qui seront utilisées pour tester les programmes des questions suivantes. Elles sont soit générées pseudo-aléatoirement, à partir d'une suite initialisée par une valeur u_0, différente pour chaque personne, distribuée au début de l'épreuve ; soit lues dans des fichiers fournis avec l'énoncé.

Nous invitons *fortement* les candidats et candidates à se familiariser à l'avance avec la manière dont ces suites pseudo-aléatoires sont générées et utilisées dans les sujets précédents afin de gagner du temps le jour de l'épreuve.

Les questions écrites demandent de calculer certaines valeurs, souvent numériques, bien que d'autres types tels que des chaînes de caractères soient également possibles. Chaque question requiert l'implémentation d'un algorithme et son utilisation sur les

entrées générées au début du sujet, pour calculer les valeurs demandées. Une question est généralement divisée en sous-questions pour des entrées de plus en plus grandes, ce qui permet de tester l'efficacité de l'algorithme mis en œuvre. Les réponses sont à inscrire sur une fiche-réponse, qui sera remise au jury à l'issue de la partie pratique de l'épreuve.

Une aide précieuse est donnée aux candidats et candidates, sous la forme d'une fiche-réponse type, contenant les valeurs obtenues sur les données générées à partir d'un $\widetilde{u_0}$ donné. Cette fiche leur permet de vérifier l'exactitude des réponses pour une graine différente de leur u_0. Il est très fortement recommandé, comme indiqué dans l'introduction des sujets, de vérifier que le générateur aléatoire se comporte comme attendu avec la graine $\widetilde{u_0}$, pour chaque question. Il serait dommage de traiter le sujet avec un générateur faux, et donc d'obtenir de mauvaises valeurs numériques malgré des algorithmes corrects.

Les questions orales sont de nature plus théorique et sont destinées à être présentées pendant la seconde partie de l'épreuve. Le déroulement de l'oral est le suivant : le candidat ou la candidate présente, le plus efficacement possible, les questions orales préparées pendant la première phase, puis éventuellement, s'il reste du temps, s'ensuit une discussion avec le jury sur les questions non traitées. La présentation orale vise à évaluer la bonne compréhension du sujet et le recul. Le jury s'efforce d'aborder toutes les questions préparées pendant la première étape, et, suivant le temps disponible, des extensions de ces questions, ou des questions qui n'ont pas été traitées par manque de temps. Pour réaliser un bon oral, il est important de prendre le temps de réfléchir aux questions à préparer mentionnées dans le sujet, et de préparer suffisamment de notes au brouillon pour être capable d'exposer clairement et efficacement les solutions, en s'aidant du tableau dans la mesure où il est utile.

La partie écrite de l'épreuve représentait cette année environ 50 % de la note finale. On observe dans l'ensemble, quoique pas systématiquement, une bonne corrélation entre les résultats obtenus aux deux parties.

Éléments de code fournis et lecture de fichiers. Cette année encore, certains sujets utilisaient des fichiers fournis : certains éléments de code peuvent être mis à disposition des candidats et candidates, et des données d'entrée sont parfois lues depuis des fichiers, plutôt que générées aléatoirement. Dans ces deux cas, les fichiers nécessaires sont distribués aux candidats et candidates au début de l'épreuve. De plus, le code permettant de lire les fichiers de données est systématiquement fourni. Il peut par exemple être demandé d'utiliser ce code pour récupérer une partie d'un fichier dépendant du u_0, qui sera utilisée comme donnée d'entrée.

Conseils et remarques générales

Écriture du programme. Le jury peut être amené à inspecter le code des candidats et candidates afin de lever certaines ambiguïtés lors de la présentation de leurs algorithmes. Cela n'est cependant possible que pour celles et ceux dont le code est suffisamment clair, avec des réponses aux questions facilement identifiables et exécutables. Nous conseillons ainsi aux candidats et candidates de soigner la lisibilité de leur code. On pourra s'inspirer des propositions de corrigés fournies en annexe de ce rapport.

Génération de structures. La plupart des sujets demandent de générer des objets (graphes, arbres, chaînes de caractères ou autre) qui seront manipulés par la suite. Bien que chaque sujet impose son propre modèle de génération de données aléatoires, certaines étapes reviennent chaque année. S'entraîner sur plusieurs sujets en conditions réelles prend un temps considérable (3h30 de préparation par sujet), ainsi nous recommandons plutôt aux candidats et candidates de traiter les premières questions de plusieurs sujets différents, afin d'être efficaces sur le début du sujet. Nous leur conseillons également de s'entraîner à traiter un ou deux sujets en entier, et de lire des corrections. Ceci leur

permettra d'avoir une idée du genre de subtilités algorithmiques qui les attendent, et de maîtriser les questions qui sont similaires d'un sujet à l'autre.

Par ailleurs, plusieurs personnes mélangent dans leur code le $\widetilde{u_0}$ commun fourni pour tester leur code, et leur u_0 propre. Pour éviter que cela ne cause de problème, nous recommandons fortement d'éviter les copier-coller (une version du code pour u_0 et une pour $\widetilde{u_0}$), et plutôt de lire le u_0 dans une variable utilisée dans tout le code, *cf.* les corrigés proposés.

Gestion de l'oral. La durée de l'oral étant courte relativement au nombre de questions à traiter, nous conseillons aux candidats et candidates de préparer une réponse précise mais intuitive, plutôt que de se perdre dans une preuve laborieuse. Si le jury n'est pas convaincu par un argument simple, il sera toujours possible de donner une preuve plus détaillée sans que cela ne diminue la note finale. Inversement, si le jury est convaincu par un raisonnement intuitif, on dispose alors de plus de temps pour la suite, permettant d'aborder des questions moins souvent traitées et donc susceptibles de rapporter beaucoup de points. Par exemple, on voit parfois des candidats ou candidates se lancer dans d'interminables preuves par induction alors qu'il existe une explication intuitive immédiate. La capacité à exposer un argument formel pour répondre à une question est évaluée dans le cadre de l'épreuve d'informatique fondamentale, tandis que l'objet de l'oral d'algorithmique est de s'assurer que le candidat ou la candidate fait le lien entre la résolution d'un problème informatique dans un cadre formel inédit et sa mise en pratique. Le jury saura donc apprécier le recul que démontre un argument simple et intuitif par rapport à une suite d'arguments formels désincarnés.

Au sujet de la gestion du tableau, nous invitons les candidats et candidates à éviter l'écueil consistant à écrire tout leur raisonnement au tableau et ainsi perdre beaucoup de temps. Le problème inverse, ne pas utiliser du tout le tableau, est plus rare, mais il rend parfois le raisonnement dur à suivre. Il est difficile de donner une règle générale sur l'utilisation du tableau, mais il ne faut pas hésiter à faire un dessin au tableau quand une explication s'y prête, puis ensuite s'appuyer dessus pour faire la preuve à l'oral. Dans le cas d'une preuve par induction, il peut être intéressant d'écrire l'hypothèse, ou un invariant, sans détailler tout le raisonnement.

Enfin, il est recommandé d'utiliser dans les réponses aux questions d'oral les mêmes notations et terminologie que dans le sujet – nous avons trop souvent vu des candidats ou candidates donner la complexité de leurs algorithmes en fonction d'un "n" non défini, ou n'ayant pas le même sens que dans le sujet. Le jury sera bien entendu capable de suivre un raisonnement qui utilise d'autres termes ou notations que le sujet, mais on risque alors de perdre du temps en explications facilement évitables.

Lecture du sujet. Nous conseillons aux candidats et candidates de lire le sujet en entier, avant de se lancer dans l'écriture de leurs programmes. Ceci peut permettre d'identifier quelles questions sont indépendantes et peuvent être traitées dans le désordre, ainsi que de voir quel genre de problèmes vont être étudiés, ce qui peut orienter le choix des structures de données.

Recherche exhaustive et solutions naïves. Il n'est pas rare que les sujets demandent pour commencer une approche naïve pour résoudre un problème sur de petites valeurs, typiquement un algorithme exhaustif, avant d'orienter les candidats et candidates vers des méthodes plus efficaces. Il est alors généralement inutile de tenter de trop optimiser les algorithmes naïfs demandés – il a certaines années semblé au jury que certaines personnes n'ont pas osé résoudre des questions par force brute, ayant correctement identifié que cela menait à une complexité exponentielle. C'est dommage, car les valeurs numériques sont alors choisies assez petites pour qu'une telle approche conclue.

Les bornes de complexité d'algorithmes par force brute ont semblé peu claires à certaines personnes : nous avons parfois vu des réponses fantaisistes donnant par exemple une

majoration du nombre de chemins dans un graphe linéaire en son nombre de sommets, et des candidats ou candidates s'étonner qu'un algorithme exponentiel ne permette pas de conclure sur de grandes valeurs.

Signalons enfin qu'un algorithme cherchant à maximiser une valeur par exploration exhaustive n'est pas la même chose qu'un algorithme glouton, comme nous avons vu certains candidats ou candidates le prétendre.

Présentation des algorithmes. Certaines questions orales demandent aux candidats et candidates de présenter leurs algorithmes et d'analyser leur complexité. Nous les encourageons vivement à le faire de façon claire et concise. Contrairement à ce que nous avons trop souvent pu voir, il ne s'agit pas de recopier un programme en Python, OCaml, ou autre au tableau. Il faut s'efforcer de présenter (uniquement) les étapes clefs de l'algorithme, en langage naturel si possible, afin de permettre efficacement l'analyse de complexité par la suite. En particulier, il est essentiel d'en identifier clairement la structure itérative ou récursive.

Quelqu'un qui propose un algorithme correct peut obtenir tous les points à l'oral même sans l'avoir implémenté durant la partie pratique de l'épreuve. Les candidats et candidates ne doivent surtout pas s'interdire d'expliquer un algorithme plus efficace que celui effectivement implémenté, qui leur serait venu à l'esprit par la suite. On peut dans ce cas expliquer d'abord l'algorithme implémenté puis comment l'améliorer, ou bien présenter directement la version optimale.

Complexité des algorithmes. Le jury décerne des points partiels aux algorithmes justes mais à la complexité non optimale. Si l'algorithme proposé est en réalité trop naïf pour traiter les instances proposées dans le sujet, le jury saura apprécier un regard critique, qui exploiterait l'analyse de complexité et un ordre de grandeur sur le nombre d'opérations élémentaires qu'un ordinateur peut effectuer. Nous n'attendons pas une estimation précise du temps de calcul, mais de savoir qu'il est difficile d'obtenir une réponse rapidement si l'algorithme demande 10^{12} tours d'une boucle.

Par ailleurs, nous souhaitons insister sur le fait que donner la complexité d'un algorithme sur une entrée donnant lieu à une exécution particulièrment lente ne constitue évidement pas une majoration de la complexité dans le pire cas de cet algorithme. Soulignons que donner explicitement le pire cas n'est pas toujours nécessaire : on peut souvent justifier une majoration de la complexité sans pour autant donner une entrée précise qui l'atteint.

Langages de programmation. En MPI, les sujets demandent explicitement d'utiliser les langages OCaml et C, sans laisser le choix, et il est donc important de s'entraîner avec les deux. Les candidats et candidates nous ont semblé cette année avoir une bonne maîtrise de OCaml, ce que nous avons apprécié, mais être moins à l'aise en C.

Les sujets de MP, quant à eux, sont prévus et calibrés pour être de difficulté équivalente dans les langages Python et OCaml. Nous notons cette année encore une tendance générale des candidats et candidates à utiliser plutôt Python que OCaml. Certaines personnes hésitent et passent du temps à chercher le "meilleur" langage pour le sujet. Ceci est une perte de temps. Il est préférable de choisir à l'avance le langage que l'on maîtrise le mieux. Cela permet de s'entraîner pour bien connaître et éviter les problèmes et limitations liées au langage. On a ainsi pu voir des candidats ou candidates se lancent en Python sans se rappeler, par exemple, que dans ce langage les appels récursifs sont limités par défaut à 1000 (*cf.* `sys.setrecursionlimit`) et que les listes sont représentées par des tableaux.

Pour finir, nous conseillons habituellement dans le rapport du jury d'étudier les structures de données basiques pour le ou les langages utilisés. Cette année, les candidats et candidates nous ont paru dans l'ensemble avoir bien compris et suivi ce conseil, ce que nous avons particulièrement apprécié. Nous le réitérons donc à l'attention des années futures. Le gain en efficacité d'un programme qui utilise une liste, un tableau, une table de hachage lorsque c'est approprié est très visible dans cette épreuve. Connaître les complexité d'un

accès, un parcours, une copie de ces structures est également souvent indispensable à l'analyse de complexité. Cela ne veut pas dire le jury s'attend à avoir tous les détails de l'implémentation lors de l'oral. Au contraire, les candidats et candidates qui obtiennent les meilleures notes savent mentionner les structures utilisées sans pour autant trop y passer de temps.

Exemple. Nous terminons par un exemple, la présentation d'un algorithme de parcours de graphe non-orienté. Voici les quatre phrases que l'on s'attend typiquement à entendre pour une telle question.

- Un algorithme de parcours de graphe part d'un sommet et suit les arêtes pour visiter les sommets du graphe connectés au sommet original.
- L'ordre de traitement des arêtes est déterminé par le choix du parcours : en largeur, on traite en priorité les arêtes par distance croissante au sommet original, et en profondeur, on traite en priorité les arêtes sortant du dernier sommet visité. Ceci induit la structure de donnée utilisée : une file (FIFO) pour un parcours en largeur, une pile (LIFO) pour un parcours en profondeur.
- Dans les deux cas, chaque arête est ajoutée dans la structure de données exactement une fois, ce qui est assuré par un tableau de booléens déterminant si un sommet a déjà été visité ou non.
- La complexité du parcours est ainsi $O(n+m)$, où n est le nombre de sommets et m le nombre d'arêtes.

Ce dernier résultat est un résultat de cours qui doit être bien intégré : un parcours bien implémenté est linéaire en la taille du graphe. L'argument clef à bien comprendre est que l'on ne parcourt chaque sommet qu'une fois et l'on ne parcourt chaque arête qu'au plus deux fois.

Sujet 1 : Lumière.

Ce sujet portait sur l'étude d'un puzzle classique, consistant en une grille carrée, dont chaque case contient une lampe qui peut être allumée ou éteinte. Appuyer sur une case inverse son état, ainsi que celui des quatre cases adjacentes. Le but est de trouver sur quelle séquence de cases appuyer pour reproduire une grille cible donnée, en partant d'une grille initialement éteinte. On voit parfois ce puzzle sous d'autres formes (équivalentes), demandant d'éteindre (ou d'allumer) toutes les cases à partir d'une grille donnée.

Le sujet se décomposait, après une partie 1 préliminaire, en trois parties indépendantes. Des candidats et candidates en ont tiré parti, par exemple en sautant au début de la partie 4. Attention cependant, il n'est pas recommandé de passer entièrement la partie en C, comme certains et certaines l'ont fait : le langage C est toujours exigé dans l'épreuve de TP, et une part non-négligeable des points y est attribuée.

Partie 1. Le sujet commençait, comme souvent, par deux questions Q1 et Q2 sur la génération des grilles cibles. La Q3 demandait ensuite de calculer la distance entre deux grilles, définie comme le nombre de cases dont l'état diffère. Ces trois questions d'échauffement ont été très bien réussies. Il fallait choisir une structure de données adaptée pour représenter les grilles (question d'oral QO1) : sans piège, un tableau à deux dimensions ou éventuellement une table de hachage faisaient l'affaire.

Partie 2. Cette partie était consacrée à la résolution du puzzle, en OCaml. La Q4 proposait tout d'abord de calculer par force brute un ensemble de cases à appuyer atteignant une distance minimale à la grille cible. Elle a été bien traitée dans l'ensemble, on a pu voir des solutions parcourant récursivement tous les ensembles de cases possibles, et des solutions itératives utilisant une fonction qui calcule le successeur d'un ensemble.

La Q5 demandait d'écrire un algorithme glouton, qui choisit à chaque étape la meilleure case possible. À la surprise du jury, seulement la moitié des candidats et candidates ont correctement implémenté cet algorithme, pourtant relativement simple. À la QO3, il

fallait pour obtenir tous les points remarquer que l'on peut maintenir à jour la distance courante, au lieu de la recalculer à chaque fois, pour obtenir une complexité en $O(n^4)$ plutôt que $O(n^6)$. Certains et certaines l'ont bien vu. La QO4 demandait si le glouton calculait toujours un résultat optimal : ce n'est pas le cas. Les candidats et candidates ne s'y sont pour la plupart pas trompés, et ont fourni des contre-exemples variés. Encore fallait-il savoir justifier que c'en étaient, par exemple en donnant le résultat renvoyé par le glouton implémenté ainsi qu'une meilleure réponse trouvée à la main.

Enfin, on donnait une modélisation du puzzle par un système d'équations linéaires sur $\mathbb{Z}/2\mathbb{Z}$. À la QO5, il fallait expliquer ce point de vue : on souhaitait principalement entendre que le choix du corps était approprié car l'appui sur une case est involutif, qu'ajouter le $i^{\text{ème}}$ vecteur colonne de la matrice proposée revenait à appuyer sur la case i, et donc que chercher un ensemble de cases atteignant une cible revenait à chercher une combinaison linéaire des colonnes égale au vecteur associé. La Q6 demandait de résoudre le système linéaire, et donc le puzzle, en implémentant l'algorithme du pivot de Gauss. Il s'agissait d'une question difficile – l'algorithme n'était pas rappelé, et son implémentation se prête à de nombreuses petites erreurs difficiles à repérer. Peu de personnes sont parvenues à écrire un programme correct. Certaines ont choisi de passer cette question pour attaquer directement la partie C. C'est une décision raisonnable, attention cependant à ne pas passer toutes les questions difficiles, qui rapportent plus de points. L'analyse de complexité du pivot, en QO6, a parfois été laborieuse, des candidats ou candidates oubliant par exemple qu'ajouter une ligne d'une matrice à une autre ne se fait pas en temps constant. D'assez nombreuses personnes avaient préparé cette question alors même que leur programme ne fonctionnait pas : c'est une très bonne idée, qui permet d'obtenir tout de même des points à l'oral. Une analyse plus fine pouvait être menée en remarquant la structure particulière de la matrice : ce n'était pas attendu, mais le jury a parfois abordé ce point avec des candidats et candidates à qui il restait du temps.

Partie 3. Cette partie, en C, s'intéressait à des matrices possédant la même structure que celle du système précédent : des matrices creuses, dont les valeurs non nulles sont réparties dans une bande de largeur ℓ autour de la diagonale. À la QO7, il fallait expliquer, sans avoir à l'implémenter, que l'on peut optimiser le stockage de telles matrices en gardant seulement les valeurs de cette bande ; on attendait une explication rapide de la relation entre les indices d'un élément dans la matrice de départ et dans la version optimisée.

Cette partie contenait trois questions relativement simples sur le plan algorithmique, mais bien sûr l'obligation d'utiliser le C est en elle-même source de difficulté. Les Q7 et Q8, demandant de générer des matrices puis de calculer la largeur de bande d'une matrice, ont été bien réussies par la plupart des personnes qui les ont abordées.

La Q9, plus difficile, a été peu réussie : elle demandait d'effectuer un parcours en largeur du graphe représenté par une matrice, pour obtenir une permutation à appliquer sur ses lignes et colonnes dans le but de réduire sa largeur de bande. Il fallait en particulier implémenter une file : comme le nombre de sommets était connu à l'avance, on pouvait simplement utiliser un tableau et deux indices pour le début et la fin de la file. Il était nécessaire, dans l'algorithme proposé, de trier les sommets par degré croissant : il fallait bien sûr remarquer qu'il suffisait d'effectuer une fois ce tri au début, plutôt qu'à chaque étape.

Partie 4. Enfin, la dernière partie du sujet, en OCaml de nouveau, était consacrée à une variante du problème, dans laquelle on veut minimiser la distance à une grille cible, avec la contrainte qu'il faut appuyer sur exactement une case sur chaque ligne de la grille. Cette partie a été peu abordée à l'écrit. La Q10 et la QO9 consistaient de nouveau à énumérer exhaustivement toutes les possibilités, sans difficulté particulière. La Q11 et la QO10, enfin, demandaient de résoudre le problème plus efficacement, par un algorithme dynamique légèrement subtil. Aucun candidat ni candidate ne les a résolues.

Écrit	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
Tous les points	100	100	98	63	45	14	55	45	8	8	0
Réponses partielles	100	100	100	71	53	14	59	49	10	10	0

Oral	QO1	QO2	QO3	QO4	QO5	QO6	QO7	QO8	QO9	QO10
Tous les points	98	84	55	73	73	39	73	14	27	0
Réponses partielles	100	100	100	90	84	78	90	41	45	10

TABLE 1. Pourcentages de réponses correctes et partielles à chaque question du sujet 1.

SUJET 2 : RÉÉCRITURE ET PARTAGE

Ce sujet aborde le problème de la réécriture de termes, représentés de façon formelle par des arbres syntaxique abstraits, ainsi que l'impact du partage de sous-termes sur la complexité en temps et en mémoire des opérations de réécriture.

La première partie du sujet, en OCaml, introduit la notion de termes abstraits, leur représentation par des arbres étiquetés, ainsi que le principe de réécriture de termes. À la fin de cette partie, les candidat·es attentifs auront remarqué qu'une répétition de N réécritures donne lieu à des termes de taille exponentielle en N. La deuxième partie du sujet, en C, demande aux candidat·es d'implémenter une structure de dictionnaire à l'aide d'une table de hachage par sondage linéaire. La complexité conceptuelle et algorithmique moindre de cette partie est compensée par les difficultés d'implémentation d'une telle structure de données dans un langage bas niveau comme le C. Enfin, la troisième et dernière partie, en OCaml, présente la notion de partage maximal (plus connue sous le nom anglais de "*hash-consing*"), et montre comment celle-ci permet de réduire drastiquement la complexité temporelle et spatiale de la réécriture, en obtenant un gain exponentiel.

Partie 1. La question Q1 demande de calculer le haché d'un terme par un simple parcours récursif. La question Q2, dans laquelle il faut implémenter une fonction de substitution, est de difficulté similaire, et se résout aussi par un simple parcours descendant. Ces deux questions, ainsi que la question orale sur la complexité d'un parcours descendant, sont toutes trois assez classiques et ont été réussi par presque tou·te·s les candidat·es. La question Q3 consiste à implémenter un algorithme de *matching*, c'est à dire à trouver une affectation de variables rendant deux termes t et p égaux (contrairement au problème d'unification, seul le terme p peut contenir des variables). Ici aussi, un simple parcours récursif descendant suffit, avec une (petite) difficulté supplémentaire : il est nécessaire de maintenir l'affectation en cours de construction, et de s'assurer que celle-ci convient en vérifiant qu'une même variable de p n'a pas besoin d'être assignée à deux sous-termes de t différents. Les candidat·es les plus pragmatiques ont réussi cette question sans efforts en stockant l'affectation dans un simple tableau global. D'autres approches ont été présentées au jury, notamment en remontant récursivement des affectations, qui doivent alors être fusionnées. Ces solutions, correctes mais sensiblement plus complexes, ont probablement ralenti certains candidat·es. Lors de l'analyse de la complexité temporelle du *matching* (question orale QO2), de nombreux candidat·es ont oublié qu'un test d'égalité structurelle entre deux termes est linéaire dans la taille des termes. La question Q4 demande d'utiliser le matching pour réécrire dans des termes. Cette question n'est pas très difficile à implémenter : la difficulté principale réside dans l'analyse théorique de celle-ci (questions orales QO3 et QO4), que peu de candidat·es ont totalement réussie. Ces deux questions demandent une analyse attentive de la taille des termes obtenus par une réécriture : notamment, tou·te·s les candidat·es n'ont pas remarqué qu'une réécriture peut doubler (ou plus) la taille d'un

terme. De façon surprenante, plus d'un tiers des candidat·es n'ont pas essayé d'aborder la question Q5. C'est dommage, car celle-ci est en réalité assez simple : il s'agit simplement de répéter la fonction réécrivant un terme implémentée en question Q4. Dans la question orale QO5 associée, elle aussi peu abordée et peu réussie, il fallait notamment observer que puisque qu'une réécriture peut multiplier la taille d'un terme par un facteur ≥ 2, la répétition de réécritures crée des termes de taille exponentielle.

Partie 2. La question Q6 demande d'implémenter en C une table de hachage par sondage linéaire. En particulier, il est seulement demandé d'implémenter la structure de données et la fonction d'ajout dans une table, ainsi qu'une fonction calculant la surcharge d'une table (cette dernière n'a pas d'intérêt pratique, et sert uniquement à évaluer les réponses des candidat·es). Il est d'abord demandé de considérer des tables de taille fixée, avant de modifier (dans la question Q7) la structure pour re-dimensionner automatiquement la taille de la table lorsque celle-ci devient trop petite. La question orale QO6, demandant de décrire les choix d'implémentation des candidat·es, a été bien réussie.

L'opération de suppression d'un élément, non triviale dans une table de hachage par sondage linéaire, est seulement abordée à l'oral dans la question QO7. Peu de candidats ont compris et géré correctement la difficulté : lors de la suppression d'un élément (k, v), il faut maintenir l'invariant que toute autre liaison (k_0, v_0) doit appartenir à un bloc contigu de liaisons contenant la cellule d'indice $\mathsf{hash}(k_0)$. Ceci explique le faible taux de réussite à QO7. Le jury note que les autres questions de cette partie (Q6, Q7, et QO6) ont été très bien traitées par les candidat·es les ayant abordées. Cependant, bien que la majorité des candidat·es aient abordé la question orale QO6, moins d'un tiers des candidat·es ont implémenté les questions écrites. Le jury se demande si cela est dû à la difficulté à le faire en C, et regrette le faible taux de réussite à l'écrit dans cette partie. Le jury rappelle que le langage C est au programme, et qu'une maîtrise élémentaire de celui-ci est attendu des candidat·es.

Partie 3. Cette partie commence par une question ayant un format inhabituel pour cette épreuve : la lecture de code. En effet, la question orale QO8 demande aux candidat·es de lire des éléments de code fournis, et d'expliquer pourquoi ceux-ci réalisent bien leur spécification telle que décrite dans l'énoncé. Ici, il faut expliquer comment une structure de table de hachage permet de construire des termes ayant la propriété de partage maximal. Notamment, il est attendu des candidat·es qu'ils observent que les fonctions fournies exploitent le *hash-consing* pour ne pas avoir à faire d'appel récursif. Ces fonctions sont donc en temps constant (à arité fixée).

Plus tard, ceci permet de justifier que la mise en forme *hash-consée* d'un terme, à implémenter dans la question Q8, est en temps linéaire. La question Q8 demande aussi d'implémenter les fonctions calculant, respectivement, la taille réelle $|t|$ et la taille en mémoire $|t|_m$ d'un terme t. La question orale QO9 s'intéresse à la complexité temporelle du calcul de $|t|_m$. Pour obtenir tous les points à cette question, il était attendu que les candidat·es observent que cela peut être fait en temps $O(|t|_m)$ et non pas $O(|t|)$, en stockant les identifiants des sous-termes déjà vus pour ne pas les ré-explorer (par exemple dans dans une liste, ou mieux dans une table de hachage). Peu de candidat·es l'ont vu.

Enfin, la question Q9 demande de ré-implémenter la réécriture de termes en exploitant la propriété de partage maximal, et la question orale QO10 d'en analyser la complexité. Pour passer les plus grand cas de Q9, il est nécessaire de mettre en place deux optimisations : il faut utiliser le fait que l'égalité structurelle de deux termes *hash-consés* peut être testée en $O(1)$; et il faut stocker les identifiants des sous-termes déjà traités. Très peu de candidat·es ont eu le temps d'aborder cette question, et aucun·e n'a réussi à la traiter entièrement.

La dernière question écrite de ce sujet (Q10) n'a été réussie par aucun·e candidat·e, probablement par manque de temps. De même, une seule personne a abordé la dernière question orale (QO11). L'écrit et l'oral de cette dernière partie ont été abordés par un très

faible nombre de candidat·es. Cela n'a pas surpris le jury, qui note que ce sujet est très long.

Écrit	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
Tous les points	98	90	86	61	42	28	17	17	0	0
Réponses partielles	100	96	94	80	63	28	19	17	5	0
Réussite moyenne	99	93	90	73	54	29	19	17	3	0

Oral	QO1	QO2	QO3	QO4	QO5	QO6	QO7	QO8	QO9	QO10	QO11
Tous les points	98	21	17	59	9	82	7	9	7	0	0
Réponses partielles	100	100	96	84	55	86	80	30	13	5	1
Réussite moyenne	99	67	67	79	32	85	43	24	13	2	1

TABLE 2. Pourcentages tronqués de réponses correctes, partielles et moyennes des points à chaque question du sujet 2.

Lumière

Épreuve pratique d'algorithmique et de programmation
Concours commun des Écoles normales supérieures

Durée de l'épreuve : 3 heures 30 minutes

Juin/Juillet 2024

ATTENTION !

N'oubliez en aucun cas de recopier votre u_0
à l'emplacement prévu sur votre fiche réponse

Important.

Il vous a été donné un numéro u_0 qui servira d'entrée à vos programmes. Les réponses attendues sont généralement courtes et doivent être données sur la fiche réponse fournie à la fin du sujet. À la fin du sujet, vous trouverez en fait deux fiches réponses. La première est un exemple des réponses attendues pour un $\widetilde{u_0}$ particulier (précisé sur cette même fiche et que nous notons avec un tilde pour éviter toute confusion !). Cette fiche est destinée à vous aider à vérifier le résultat de vos programmes en les testant avec $\widetilde{u_0}$ au lieu de u_0. Vous indiquerez vos réponses (correspondant à votre u_0) sur la seconde et vous la remettrez à l'examinateur à la fin de l'épreuve.

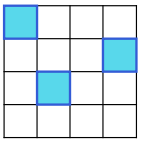
En ce qui concerne la partie orale de l'examen, lorsque la description d'un algorithme est demandée, vous devez présenter son fonctionnement de façon schématique, courte et précise. Vous ne devez en aucun cas recopier le code de vos procédures !
Quand on demande la complexité en temps ou en mémoire d'un algorithme en fonction d'un paramètre n, on demande l'ordre de grandeur en fonction du paramètre, par exemple : $O(n^2)$, $O(n \log n)$,...
Il est recommandé de commencer par lancer vos programmes sur de petites valeurs des paramètres et de ***tester vos programmes sur des petits exemples que vous aurez résolus préalablement à la main ou bien à l'aide de la fiche réponse type fournie en annexe***. Enfin, il est recommandé de lire l'intégralité du sujet avant de commencer afin d'effectuer les bons choix de structures de données dès le début.

Les parties 1, 2 et 4 doivent être traitées en OCaml, et la partie 3 en C. Ces consignes de langage seront répétées au début des parties concernées. Elles doivent impérativement être suivies.
Pour la partie 3, un fichier `base.c` est fourni. Il contient quelques fonctions prédéfinies dont vous pouvez vous servir si vous le souhaitez. Il est fortement conseillé de faire une copie de ce fichier, pour ne pas risquer de l'effacer par erreur.
La partie 1 présente le problème étudié, ainsi que la génération des données qui seront utilisées dans les parties 2 et 4, et doit donc être traitée avant ces dernières. Les parties 2, 3 et 4 sont indépendantes.

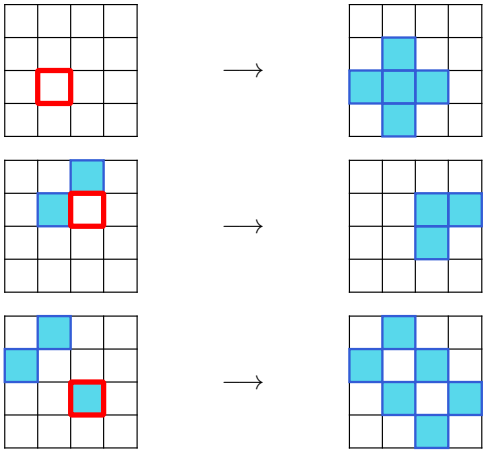
1 Préliminaires

1.1 Un problème d'éclairage

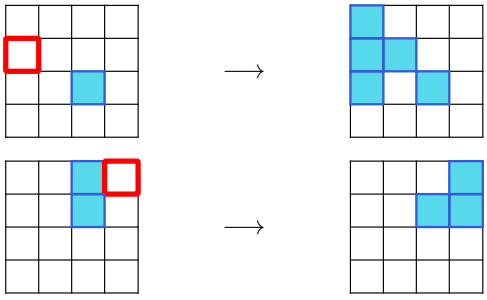
On s'intéresse à un puzzle qui se joue sur une grille carrée de taille $n \times n$, pour un entier $n \geq 1$. Chaque case de la grille est une lampe, qui a deux états possibles : elle peut être allumée ou bien éteinte. On peut voir ci-dessous l'exemple d'une grille 4×4, dans laquelle trois cases sont allumées.



Une instance du puzzle est une grille donnée, dont certaines cases sont allumées et d'autres éteintes. Le but du jeu est d'allumer les lampes de façon à reproduire cette grille, en partant d'une grille initialement complètement éteinte. On appellera souvent dans la suite cette grille à atteindre la *cible*.
Pour cela, la seule action autorisée est d'appuyer sur les cases de la grille. Appuyer sur une case a pour effet d'inverser son état, ainsi que celui des quatre cases directement adjacentes, au dessus, en dessous, à gauche, et à droite : une case éteinte s'éclaire, une case allumée s'éteint. Dans les trois exemples ci-dessous, si l'on appuie sur la case marquée dans la grille de gauche, on obtient la grille de droite.

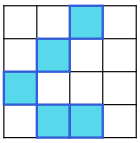


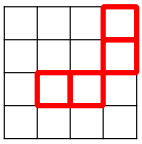
Appuyer sur une case modifie ainsi l'état d'au plus cinq cases, disposées en forme de croix. Si la case sur laquelle on appuie est située au bord de la grille, cette croix sera tronquée, comme on peut le voir dans les deux exemples suivants.



Résoudre le puzzle consiste à trouver une séquence de cases sur lesquelles appuyer pour reproduire la grille cible. Notons qu'il n'y a *a priori* en général ni existence ni unicité de la solution.

On remarque rapidement qu'appuyer plusieurs fois sur la même case est inutile, et que l'ordre dans lequel les cases sont choisies n'a pas d'importance. Plutôt qu'une séquence ordonnée, on recherchera par conséquent un ensemble de cases qui permettront d'obtenir la grille voulue.
Par exemple, si la grille cible est celle ci-dessous à gauche, l'ensemble des cases marquées sur la grille de droite est une solution : appuyer une fois sur chacune de ces cases, dans n'importe quel ordre, produira la grille voulue.

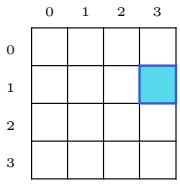




1.2 Notations

On rappelle que, pour tous $a, b \in \mathbb{Z}$ tels que $b \neq 0$, $a \bmod b$ désigne le reste de la division euclidienne de a par b, c'est-à-dire le plus petit entier naturel r tel qu'il existe un entier q vérifiant $a = b \times q + r$. Par ailleurs, pour un entier naturel n, on note $[\![0, n]\!]$ l'ensemble des entiers jusqu'à n : $\{0, 1, 2, \ldots, n-1, n\}$.

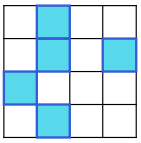
Dans une grille G de taille $n \times n$, on identifiera chaque case avec sa position, c'est-à-dire le couple d'entiers $(i, j) \in [\![0, n-1]\!]^2$ formé de sa ligne et sa colonne, en commençant en haut à gauche de G.
Par exemple, la case $(1, 3)$ est allumée dans la grille ci-dessous.

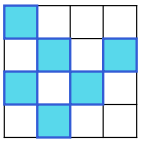


On note $\mathcal{A}(\mathsf{G})$ l'ensemble des cases allumées de G. Pour deux grilles G, G' de même taille, la *distance* de G à G', notée $\mathsf{d}(\mathsf{G}, \mathsf{G}')$, est définie comme le nombre de cases n'ayant pas le même état dans les deux grilles :

$$\mathsf{d}(\mathsf{G}, \mathsf{G}') = \Big| \big(\mathcal{A}(\mathsf{G}) \setminus \mathcal{A}(\mathsf{G}')\big) \ \cup \ \big(\mathcal{A}(\mathsf{G}') \setminus \mathcal{A}(\mathsf{G})\big) \Big|$$

Par exemple, les deux grilles ci-dessous sont à distance 3.





Pour un entier $n \geq 1$ et un ensemble de cases $\mathsf{C} \subset [\![0, n-1]\!]^2$, on appellera l'image de C, notée $\mathcal{I}(\mathsf{C})$, la grille obtenue en appuyant une fois sur chaque case de C, en partant d'une grille vide.

Pour tout entier $n \geq 1$, on considère l'ordre lexicographique sur les cases de $[\![0, n-1]\!]^2$: $(i, j) \leq (i', j')$ lorsque $i < i'$, ou $i = i' \ \wedge \ j \leq j'$.

1.3 Génération de données

Les trois questions qui suivent doivent être traitées en OCaml.
Étant donné u_0, on définit par récurrence :

$$\forall i \in \mathbb{N}.\ u_{i+1} = (569 \times u_i) \bmod 2\,424\,259$$

Question 1 *Écrire un programme qui calcule la suite u, et en donner les valeurs suivantes :*

a) $u_1 \bmod 10\,000$ ***b)*** $u_{128} \bmod 10\,000$ ***c)*** $u_{2\,024} \bmod 10\,000$ ***d)*** $u_{123\,456} \bmod 10\,000$

Indication. Il pourra être judicieux, afin d'obtenir de meilleures performances dans les questions suivantes, de précalculer les valeurs de u_n jusqu'à un n assez grand : on pourra les calculer au début du programme, et les mémoriser dans un tableau, pour ne pas devoir refaire le calcul à chaque fois.

```
let max_n : int = 2000000
let un_tab : int array = Array.make (max_n + 1) 0
let f u = (569 * u) mod 2424259

let _ =
  un_tab.(0) <- u0;
  for i = 1 to max_n do
    un_tab.(i) <- f un_tab.(i-1);
  done

let un (n:int) : int =
  assert (n <= max_n);
  un_tab.(n)

let q1 (n:int) : int =
  (un n) mod 10000
```

Pour tous entiers naturels n, m, p, avec $n > 0$ et $m \leq p$, on notera $\mathsf{G}_{n,m,p}$ la grille de taille $n \times n$ dont les cases qui sont allumées sont celles aux positions (i, j) telles que $(u_{n^2+89m+71p+ni+j} \bmod p) < m$.

Question 2 *Écrire un programme qui calcule $\mathsf{G}_{n,m,p}$. Donner, pour les entrées suivantes, la somme :*

$$\left(\sum_{(i,j) \in \mathcal{A}(\mathsf{G}_{n,m,p})} i + j \right) \bmod 10\,000.$$

a) $n = 3, m = 1, p = 4$ ***b)*** $n = 10, m = 3, p = 4$ ***c)*** $n = 100, m = 2, p = 5$

```
(* représentation de la grille par une matrice de booléens *)
type case = int * int
type grille = bool array array
```

```
(* quelques fonctions pour créeer une grille vide,
   obtenir la taille d'une grille, lire ou inverser l'état d'une case,
   lister les cases allumées *)
let vide (n:int) : grille = Array.make_matrix n n false
let taille (g:grille) : int = Array.length g
let est_allume (g:grille) ((i, j):case) : bool = g.(i).(j)
let inverse (g:grille) ((i, j):case) : unit = g.(i).(j) <- not (g.(i).(j))
let cases_allumees (g:grille) : case list =
  let l = ref [] in
  Array.iteri
    (fun i -> Array.iteri (fun j b -> if b then l := (i,j) :: !l)) g;
  !l

(* appuie sur la case p de g (sera utile plus tard) *)
let appuie (g:grille) (p:case) : unit =
  let n = taille g in
  let (i,j) = p in
  let c = [i, j; i, j+1;  i-1, j; i, j-1; i+1, j] in
  let c = List.filter (fun (x,y) -> x < n && y < n && x >= 0 && y >= 0) c in
  List.iter (inverse g) c

(* calcul de la somme pour les valeurs numériques demandées *)
let somme_ensemble (c:case list) : int =
  let s = List.fold_left (fun s (i,j) -> s + i + j) 0 c in
  s mod 10000


(* calcul de G_n,m,p *)
let gnmp (n:int) (m:int) (p:int) : grille =
  let g = vide n in
  for i=0 to n-1 do
    for j=0 to n-1 do
      let pp = (un (n*n + 89*m + 71*p + n*i + j)) mod p in
      if pp < m then
        inverse g (i,j);
    done;
  done;
  g

let q2 (n:int) (m:int) (p:int) : int =
  let g = gnmp n m p in
  somme_ensemble (cases_allumees g)
```

Question à développer pendant l'oral 1 *Décrire la structure de données utilisée pour représenter les grilles. Évaluer sa complexité en espace, ainsi que la complexité en temps des opérations consistant à appuyer sur une case et à parcourir la liste des cases allumées.*

Un choix naturel est de représenter une grille G de taille $n \times n$ par un tableau de booléens t à deux dimensions, de taille $n \times n$, en stockant dans $t[i][j]$ un booléen indiquant si la case (i, j) est allumée.

Ceci occupe un espace $\mathcal{O}(n^2)$. Appuyer sur une case demande de modifier au plus cinq cases de `t`, et a donc une complexité temporelle en $\mathcal{O}(1)$. Parcourir la liste des cases allumées demande de parcourir tout le tableau, et a donc une complexité en temps $\mathcal{O}(n^2)$.

On peut envisager d'autres codages, par exemple stocker simplement l'ensemble des cases allumées, dans une table de hachage : dans ce cas, l'espace occupé est $\mathcal{O}(|\mathcal{A}(\mathsf{G})|)$, l'appui sur une case est en $\mathcal{O}(1)$, le

parcours des cases allumées est en $\mathcal{O}(|\mathcal{A}(\mathsf{G})|)$. C'est plus efficace si peu de cases sont allumées, mais ce n'était pas nécessaire pour pouvoir répondre aux questions suivantes.

Question 3 *Écrire un programme qui calcule la distance* d. *Donner les distances suivantes.*

a) $\mathsf{d}(\mathsf{G}_{3,1,4}, \mathsf{G}_{3,1,2})$ ***b)*** $\mathsf{d}(\mathsf{G}_{10,3,4}, \mathsf{G}_{10,1,2})$ ***c)*** $\mathsf{d}(\mathsf{G}_{100,2,5}, \mathsf{G}_{100,3,7})$

```
let distance (g:grille) (h:grille) : int =
  let n = taille g in
  let d = ref 0 in
  for i=0 to n-1 do
    for j=0 to n-1 do
      if est_allume g (i,j) <> est_allume h (i,j) then
        incr d;
    done;
  done;
  !d

let q3 (n:int) (m:int) (p:int) (m':int) (p':int) : int =
  let g = gnmp n m p in
  let h = gnmp n m' p' in
  distance g h
```

2 Résolution du puzzle

Cette partie doit être traitée en OCaml.

Cette partie est consacrée à la résolution du puzzle décrit dans la partie précédente.

2.1 Recherche exhaustive

On se propose, dans un premier temps, d'adopter une approche exhaustive, consistant à essayer un par un tous les ensembles de cases possibles, jusqu'à trouver une solution. Plus précisément, puisqu'il n'existe *a priori* pas forcément de solution, on recherchera dans cette question un ensemble de cases C qui permet de se rapprocher le plus possible de la grille cible G, c'est-à-dire tel que $\mathsf{d}(\mathsf{G}, \mathcal{J}(\mathsf{C}))$ est minimal.
Si plusieurs ensembles de cases peuvent convenir, on en renverra un qui rend maximale la valeur

$$\left(\sum_{(i,j)\in\mathsf{C}} i+j\right) \bmod 10\,000.$$

Question 4 *Écrire un programme qui effectue la recherche exhaustive décrite ci-dessus. Pour chacune des grilles suivantes, calculer pour l'ensemble de cases* C *obtenu la valeur*

$$\left(\sum_{(i,j)\in\mathsf{C}} i+j\right) \bmod 10\,000.$$

a) $\mathsf{G}_{3,1,2}$ ***b)*** $\mathsf{G}_{4,1,4}$ ***c)*** $\mathsf{G}_{4,3,4}$

```
(*  variation de distance g h lorsque l'on appuie sur la case p de h *)
let delta (g:grille) (h:grille) (p:case) : int =
  let n = taille g in
  assert (n = taille h);
  let (i,j) = p in
  let c = [i, j; i, j+1;  i-1, j; i, j-1; i+1, j] in
  let c = List.filter (fun (x,y) -> x < n && y < n && x >= 0 && y >= 0) c in
  List.fold_left
    (fun x p ->
      x + (if (est_allume g p) = (est_allume h p) then 1 else -1))
    0 c


let forcebrute (n:int) (g:grille) : case list =
  let h = vide n in (* grille de travail *)

  (* calcule le meilleur choix possible pour les cases <= p,
     sachant que c est le choix actuel pour les cases > p,
     qu'il atteint une distance d et une somme s,
     et que le meilleur choix jusque là est cmin qui atteint dmin, smax *)
  let rec aux p c d s cmin dmin smax =
    let (i,j) = p in
    if i = -1 then (* plus de case à traiter :
                      on regarde si on a trouvé un meilleur ensemble *)
      (if d < dmin || (d = dmin && s > smax) then (c, d, s)
       else (cmin, dmin, smax))
    else
      let pred = (* prédécesseur de p pour l'ordre lexicographique *)
        if j = 0 then (i-1, n-1) else (i, j-1) in
      (* on essaie sans appuyer sur p *)
      let c1, d1, s1 = aux pred c d s cmin dmin smax in

      (* si on appuie sur p : la distance et la somme actuelles changent *)
      let dd = delta g h p in
      let ds = i + j in
      appuie h p;
      let c2, d2, s2 = aux pred (p::c) (d+dd) ((s+ds) mod 10000) c1 d1 s1 in
      appuie h p;
      (c2, d2, s2)
  in

  (* on appelle aux en partant d'une grille vide et de la plus grande case *)
  let d = distance g h in
  let c, _, _ = aux (n-1,n-1) [] d 0 [] d 0 in
  c

let q4 (n:int) (m:int) (p:int) : int =
  let g = gnmp n m p in
  let c = forcebrute n g in
  somme_ensemble c
```

Question à développer pendant l'oral 2 *Décrire le fonctionnement de votre programme. Évaluer sa complexité en temps.*

On peut énumérer tous les ensembles de cases possibles de façon récursive. On considère la dernière case, $(n-1, n-1)$: on peut soit appuyer soit ne pas appuyer sur cette case. On traite donc chacune de ces

deux possibilités, avec un appel récursif pour examiner le cas de la case précédente, et ainsi de suite. On s'arrête lorsque l'on atteint la première case, $(0, 0)$: on a alors fait un choix pour chaque case, et on peut calculer la distance atteinte par l'ensemble C résultant. On retient le meilleur ensemble vu, pour le renvoyer à la fin.

En procédant ainsi, on examinera une fois chacun des 2^{n^2} ensembles de cases possibles, pour lequel on calculera une distance ($\mathcal{O}(n^2)$ opérations, puisque l'on parcourt chaque case de la grille). On obtient donc une complexité en temps en $\mathcal{O}(n^2 \cdot 2^{n^2})$.

La solution proposée en corrigé est légèrement plus efficace : au lieu de recalculer entièrement la distance pour chaque chemin, elle tient à jour la distance atteinte par l'ensemble de cases courant au fil de l'exécution. Mettre à jour cette distance lorsque l'on appuie sur une case (fonction `delta`) se fait en temps constant (on ne doit considérer que cinq cases au plus). On obtient ainsi une complexité en $\mathcal{O}(2^{n^2})$.

2.2 Algorithme glouton

Pour obtenir un programme plus rapide, on propose à présent de recourir à un algorithme glouton.

Il s'agit de choisir successivement à chaque étape une case non encore utilisée qui nous rapproche le plus possible de la cible – ou nous en éloigne le moins. À une étape donnée, si C est l'ensemble des cases déjà sélectionnées, on choisira donc une case $c \notin \mathsf{C}$, qui rend $\mathsf{d}(\mathsf{G}, \mathcal{J}(\mathsf{C} \cup \{c\}))$ aussi petit que possible, pour l'ajouter à C. Lorsque l'on a le choix, on prendra toujours la case c la plus grande pour l'ordre lexicographique. On s'arrête lorsque toutes les cases ont été sélectionnées.

Au cours de l'algorithme, C se voit donc progressivement augmenté case par case, jusqu'à toutes les contenir. À la fin, on examine les valeurs successives de l'ensemble C au cours de l'exécution. On renvoie celle qui avait permis d'obtenir la distance $\mathsf{d}(\mathsf{G}, \mathcal{J}(\mathsf{C}))$ la plus petite. Si l'on a le choix, on renverra la première telle valeur de C, c'est-à-dire celle qui contient le moins de cases.

Question 5 *Écrire un programme qui applique l'algorithme glouton décrit ci-dessus. Pour chacune des grilles suivantes, calculer l'ensemble de cases C obtenu, et donner la valeur*

$$\left(\sum_{(i,j) \in \mathsf{C}} i + j \right) \bmod 10\,000.$$

a) $\mathsf{G}_{3,3,4}$ ***b)*** $\mathsf{G}_{20,2,5}$ ***c)*** $\mathsf{G}_{40,5,7}$

```
let glouton (g:grille) : case list =
  let n = taille g in
  let h = vide n in (* grille de travail *)
  let d, c = ref (distance h g), ref [] in (* distance et c courants *)
  let dmin, cmin = ref !d, ref !c in (* meilleurs distance et c vus *)

  (* table pour stocker les cases choisies. *)
  let deja_pris = Array.make_matrix n n false in
```

```
for k = 1 to n * n do (* on va ajouter toutes les cases -> n^2 tours *)
    (* on parcourt toutes les cases pour trouver la meilleure *)
    let ddmin = ref 6 in (* le plus bas delta trouvé jusque là *)
    let pmin = ref (n,n) in (* la case qui le réalise *)
    for i=0 to n-1 do
      for j=0 to n-1 do
        let dd = delta g h (i, j) in
        if dd <= !ddmin && not deja_pris.(i).(j) then (* trouvé mieux *)
          (ddmin := dd; pmin := (i, j))
      done;
    done;

    (* on choisit la meilleure case trouvée *)
    d := !d + !ddmin;
    c := !pmin :: !c;
    deja_pris.(fst !pmin).(snd !pmin) <- true;
    appuie h !pmin;

    (* on met à jour la meilleure distance vue, si nécessaire *)
    if !d < !dmin then (dmin := !d; cmin := !c);
  done;
  !cmin

let q5 (n:int) (m:int) (p:int) : int =
  let g = gnmp n m p in
  let c = glouton g in
  somme_ensemble c
```

Question à développer pendant l'oral 3 *Décrire le fonctionnement de votre programme. Évaluer sa complexité en temps et en espace.*

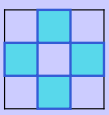
Le programme proposé ci-dessus suit directement la description de l'algorithme glouton. Un point notable est l'utilisation d'une table pour stocker les cases déjà choisies – d'autres choix sont possibles, mais il faut faire attention à effectuer le test d'appartenance en temps constant. Notons aussi qu'on réutilise la fonction `delta`, pour calculer en temps constant la variation de distance causée par l'appui sur une case, plutôt que de recalculer entièrement la distance (ce qui serait en $\mathcal{O}(n^2)$).
Pour ce qui est de la complexité en temps :

- le calcul initial de distance, et la création de la grille de travail `h` sont en $\mathcal{O}(n^2)$;
- la boucle principale est exécutée n^2 fois, puisque l'on choisit une case à chaque tour ;
- à chaque tour de boucle, la recherche de la meilleure case est en $\mathcal{O}(n^2)$ (n^2 appels à `delta` et tests d'appartenance), et son ajout à la table en $\mathcal{O}(1)$.

Au total, on obtient une complexité en $\mathcal{O}(n^4)$.

Question à développer pendant l'oral 4 *L'algorithme glouton calcule-t-il toujours une solution, si elle existe ? Si oui, justifier, si non, donner un contre-exemple.*

L'algorithme glouton ne calcule pas toujours une solution au puzzle, lorsqu'il en existe une. Par exemple, pour $n = 3$, si la cible est la grille suivante :



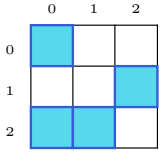
Cette instance du puzzle a une solution (qui est unique), consistant à appuyer une fois sur chacune des quatre cases allumées. L'algorithme glouton va cependant sélectionner à la première étape la case centrale $(1,1)$, qui, à partir d'une grille vide (distance 4) permet de diminuer de 3 la distance. Après avoir appuyé sur $(1,1)$, on obtient une grille à distance 1. La case $(1,1)$ sera donc nécessairement présente dans l'ensemble renvoyé par l'algorithme glouton. En l'exécutant jusqu'au bout, on constate qu'il renvoie en fait $\{(1,1)\}$, qui n'est pas une solution du puzzle.

2.3 Algèbre linéaire

Ce puzzle peut être résolu efficacement en appliquant des techniques d'algèbre linéaire, permettant de déterminer si une solution existe, et le cas échéant de la calculer.
Pour cela, on modélise le puzzle par un système d'équations linéaires sur le corps $\mathbb{Z}/2\mathbb{Z}$.
Considérons une grille cible G de taille $n \times n$. On posera $N = n^2$. G peut être vue comme un vecteur colonne $\mathsf{V_G} = (g_k)_{0 \leq k \leq N-1}$ dans $(\mathbb{Z}/2\mathbb{Z})^N$, dont la $(ni+j)$-ième composante vaut 1 si la case (i,j) de G est allumée, et 0 sinon. Autrement dit :

$$\forall (i,j) \in [\![0, n-1]\!]^2.\ g_{ni+j} = 1 \ \Leftrightarrow\ (i,j) \in \mathcal{A}(\mathsf{G}).$$

Par exemple, la grille ci-dessous à gauche sera représentée par le vecteur à droite.

$\mathsf{G} =$ 

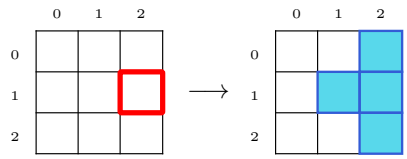
$$\mathsf{V_G} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

De la même manière, on peut voir un ensemble de cases C comme le vecteur colonne $\mathsf{V_C} = (c_k)_{0 \leq k \leq N-1}$ dont la $(ni+j)$-ième composante vaut 1 lorsque la case (i,j) est dans C :

$$\forall (i,j) \in [\![0, n-1]\!]^2.\ c_{ni+j} = 1 \ \Leftrightarrow\ (i,j) \in \mathsf{C}.$$

On construit également une matrice de taille $N \times N$, $\mathsf{M} = (m_{k,l})_{0 \leq k,l \leq N-1}$, telle que pour tous entiers naturels i, j, i', j' inférieurs à $n-1$, $m_{ni+j,ni'+j'}$ vaut 1 si et seulement si l'état de la case (i,j) dans G s'inverse lorsque l'on appuie sur la case (i',j').
Par exemple pour $n = 3$, M est la matrice 9×9 suivante :

$$\mathsf{M} = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & \mathbf{0} & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$



Comme on le voit à droite, la case $(0,2)$ change d'état lorsque l'on appuie sur $(1,2)$, et par conséquent $m_{2,5} = 1$. En revanche, la case $(2,0)$ ne change pas d'état, et par conséquent $m_{6,5} = 0$.

Résoudre le puzzle revient à résoudre le système linéaire de N équations suivant, où X est un vecteur colonne de N inconnues :

$$\mathsf{M} \cdot X = \mathsf{V}_\mathsf{G}.$$

En effet, pour tout ensemble de cases C, on peut montrer (et on admettra dans la suite) que

$$\mathsf{M} \cdot \mathsf{V}_\mathsf{C} = \mathsf{V}_\mathsf{G} \;\Leftrightarrow\; \mathcal{J}(\mathsf{C}) = \mathsf{G}.$$

Question à développer pendant l'oral 5 *Justifier le résultat précédent : pourquoi les solutions du puzzle et du système linéaire sont-elles les mêmes ? On n'attend pas une démonstration détaillée, mais une explication informelle du lien entre les deux.*

Appelons $(e_k)_{0\leq k<n^2}$ les n^2 vecteurs de $(\mathbb{Z}/2\mathbb{Z})^{n^2}$ que sont les colonnes de la matrice M.
En partant d'une grille G, appuyer sur la case (i,j) de G inverse l'état de cette case et des (au plus) quatre cases adjacentes. En reportant cela sur le vecteur V_G, cela a pour effet d'ajouter 1 à la composante correspondant à chacune des cases concernées ($ni+j$ pour (i,j), *etc.*). Par construction de M, cela revient à ajouter e_{ni+j} à V_G. Il est ici important d'avoir pris des vecteurs sur $\mathbb{Z}/2\mathbb{Z}$, puisqu'inverser l'état d'une case allumée doit la faire passer à 0 de nouveau.
Pour un ensemble C, l'image $\mathcal{J}(\mathsf{C})$ est obtenue en appuyant sur chaque case de C, et le vecteur associé à cette grille est donc la somme des e_{ni+j} pour $(i,j) \in \mathsf{C}$.
Chercher un ensemble dont l'image soit G revient donc à trouver une combinaison linéaire des (e_k) qui donne V_G : c'est ce qu'exprime le système linéaire avec M.

Il ne reste donc plus qu'à résoudre ce système linéaire. Ceci peut être fait grâce à la méthode bien connue du pivot de Gauss.

Question 6 *Écrire un programme qui utilise la méthode du pivot de Gauss pour résoudre le puzzle. Pour chacune des grilles G suivantes, calculer une solution C du puzzle, et donner la somme*

$$\left(\sum_{(i,j)\in\mathsf{C}} i+j\right) \bmod 10\,000.$$

a) $\mathsf{G}_{6,4,7}$ ***b)*** $\mathsf{G}_{31,4,7}$ ***c)*** $\mathsf{G}_{43,4,7}$ ***d)*** $\mathsf{G}_{52,4,7}$

Indication. Pour les valeurs numériques demandées, le puzzle a toujours une et une seule solution.

```
(* calcul du vecteur V_G *)
let vg (n:int) (g:grille) : bool array =
  let v = Array.make (n*n) false in
  List.iter (fun (i,j) -> v.(i*n + j) <- true) (cases_allumees g);
  v

(* calcul de la matrice M *)
let matrice_lumiere (n:int) : bool array array =
  let nn = n * n in
  let m = Array.make nn (Array.make 0 false) in
  for i = 0 to nn - 1 do
    m.(i) <- Array.make nn false;
  done;
  for i=0 to (n-1) do
    for j=0 to (n-1) do
      let l = [(i, j); (i, j+1); (i-1, j); (i, j-1); (i+1, j)] in
      let l = List.filter
                (fun (x,y) -> x >= 0 && y >= 0 && x < n && y < n) l
      in
      List.iter (fun (x,y) -> m.(i * n + j).(x * n + y) <- true) l;
    done;
  done;
  m

(* calcul de C à partir de V_C *)
let cases_vc (n:int) (vc:bool array) : case list =
  let l = ref [] in
  Array.iteri (fun p vcp -> if vcp then l := (p / n, p mod n):: !l) vc;
  !l

(* addition dans Z/2Z *)
let xor (a:bool) (b:bool) : bool = a <> b

(* ajoute la ligne i2 à la ligne i1 à la fois dans a et b *)
let ajoute a b i1 i2 =
  Array.iteri (fun j x -> a.(i1).(j) <- xor x a.(i1).(j)) a.(i2);
  b.(i1) <- xor b.(i2) b.(i1)

(* renvoie une copie d'une matrice carrée a *)
let copie a =
  let n = Array.length a in
  let aa = Array.make_matrix n 0 false in
  Array.iteri (fun i ai -> (aa.(i) <- Array.copy ai)) a;
  aa

(* échange les lignes i1 et i2 dans a *)
let echange a i1 i2 =
  let c = a.(i1) in
  a.(i1)<- a.(i2);
  a.(i2)<- c
```

```
(* pivot de Gauss *)
(* a matrice n * n, b vecteur n, résout a x = b *)
(* dans le cas où a est de rang n *)
let pivot (n:int) (a:bool array array) (b:bool array) =
  (* copies de a, b pour ne pas toucher aux originaux *)
  let a = copie a in
  let b = Array.copy b in

  (* a est supposée de rang n : on sait qu'on trouvera un pivot dans chaque colonne
  ↪ *)
  for j=0 to n - 1 do
    let p = ref j in (* on cherche un pivot non nul dans la colonne j, sous la ligne
    ↪ j *)
    while !p < n && not (a.(!p).(j)) do
      incr p;
    done;
    (*  on place cette ligne à la ligne j,
        puis on l'ajoute aux autres si nécessaire *)
    echange a !p j;
    echange b !p j;
    for i = j+1 to (n-1) do
      (* il n'est utile d'ajouter la ligne j à la ligne i que lorsque a_i,j est non
      ↪ nul *)
      if a.(i).(j) then ajoute a b i j;
    done;
  done;

  (* la matrice est maintenant sous forme échelonnée *)
  (* on remonte les colonnes pour calculer la solution x *)
  let x = Array.make n false in
  for j = n - 1 downto 0 do
    x.(j) <- b.(j); (* la solution est imposée par b *)
    if x.(j) then (* on propage aux lignes d'au dessus si besoin *)
      for i=0 to j - 1 do
        if a.(i).(j) then
          b.(i) <- not b.(i);
      done;
  done;
  x

(* résolution du puzzle *)
let resout (g:grille) : case list =
  let n = taille g in
  let a = matrice_lumiere n in
  let b = vg n g in
  let x = pivot (n * n) a b in
  cases_vc n x


let q6 (n:int) (m:int) (p:int) : int =
  let g = gnmp n m p in
  let c = resout g in
  somme_ensemble c
```

Question à développer pendant l'oral 6

Expliquer le fonctionnement de votre programme. Évaluer sa complexité en temps.

Le programme proposé contient, en plus du pivot de Gauss lui-même, plusieurs opérations.

- Des fonctions qui calculent le vecteur associé à une grille, et l'ensemble de cases codé par un vecteur, qui procèdent toutes deux en parcourant une fois la grille/le vecteur (de taille $N = n^2$), et ont donc une complexité temporelle en $\mathcal{O}(n^2)$.
- Une fonction qui génère la matrice M, en parcourant tous les (i, j) pour modifier, pour chacun, au plus cinq cases de la matrice : $\mathcal{O}(n^2)$ opérations pour le calcul, mais $\mathcal{O}(n^4)$ en comptant l'initialisation de M.
- Une fonction qui ajoute une ligne de la matrice à une autre : un parcours de la ligne, en $\mathcal{O}(n^2)$.
- Une fonction qui échange deux lignes de la matrice : $\mathcal{O}(1)$.

La fonction `pivot` est la plus importante : elle implémente l'algorithme du pivot de Gauss, dans le cas restreint où le système a une solution unique, c'est-à-dire lorsque M est de rang n^2. L'indication nous garantit qu'il suffit de traiter ce cas.
On sait donc qu'on trouvera n^2 pivots : exactement un dans chaque ligne et chaque colonne de la matrice M. On parcourt donc les colonnes de M. Dans la colonne j, on cherche un pivot, c'est-à-dire un coefficient non nul de M, sur l'une des lignes en dessous de la j-ième (on a déjà traité les lignes au dessus). On utilise donc une boucle qui parcourt au plus les $\mathcal{O}(n^2)$ lignes. Une fois le pivot trouvé, on échange les lignes pour le placer en ligne j : $\mathcal{O}(1)$ opérations. On parcourt ensuite les $\mathcal{O}(n^2)$ lignes en dessous de j : on leur ajoute si besoin la ligne du pivot, pour annuler leur j-ième coefficient. Chaque ajout est en $\mathcal{O}(n^2)$: $\mathcal{O}(n^4)$ en tout pour cette boucle.
Une fois les n^2 pivots trouvés on a donc fait $\mathcal{O}(n^6)$ opérations, et la matrice est sous forme échelonnée. Dans le cas général, il faudrait vérifier que le système obtenu est cohérent et a bien une solution, mais ici on le suppose. On n'a plus qu'à remonter en partant de la dernière colonne pour calculer la solution. Pour chaque colonne j, on lit dans le vecteur `b` la valeur de la j-ième inconnue, et si besoin on la propage aux lignes d'au dessus ($\mathcal{O}(n^2)$ opérations). Cette dernière phase est donc en $\mathcal{O}(n^4)$.
En tout, la fonction `pivot` est donc de complexité temporelle $\mathcal{O}(n^6)$. C'est aussi la complexité de la fonction finale, qui traduit le problème en système linéaire, le résout, puis fait la traduction de la solution vers l'ensemble de cases.

On peut, sans changer le programme, mener une analyse de complexité plus fine, qui exploite la forme de la matrice. En effet, comme on le voit dans la partie qui suit, la matrice M a une largeur de bande de n. On peut observer que cette propriété est (du moins pour les lignes sous les pivots déjà choisis) préservée pendant l'exécution de l'algorithme du pivot. Par conséquent, dans la boucle principale de la fonction `pivot`, lorsque l'on parcourt les $\mathcal{O}(n^2)$ lignes restantes pour leur ajouter la ligne du pivot, on sait qu'en réalité seules $\mathcal{O}(n)$ d'entre elles ont un coefficient non nul. À condition qu'on ait pensé à ne faire l'ajout de ligne que lorsque c'est utile, la phase d'ajout des lignes est donc en réalité en $\mathcal{O}(n^3)$, ce qui donne en fin de compte $\mathcal{O}(n^5)$ pour tout le programme.

3 Matrices en C

Cette partie doit être traitée en C.
À la fin de la partie 2, on a été amené à représenter le problème par un système d'équations linéaires, dont la résolution permettait de trouver une solution au puzzle.
La matrice associée à ce système linéaire avait une structure particulière : ses coefficients étaient majoritairement nuls, et les coefficients non nuls étaient ramassés autour de la diagonale. On peut le voir par exemple dans la matrice 9×9 donnée en 2.3.

Les matrices de ce type, que l'on appelle parfois *matrices bandes*, possèdent des propriétés intéressantes en termes algorithmiques : certains algorithmes peuvent tirer parti de leur structure, pour être plus efficaces que dans le cas général.

On étudiera, dans cette partie, la représentation de matrices bandes en C. On considèrera des matrices dont les coefficients sont 0 ou 1.

3.1 Données de test

Pour le test numérique des programmes de cette partie, on utilisera de nouveau des données générées à partir de la suite u. On définit pour tous entiers naturels n, m, p, avec $n \geq 1$ et $m \leq p$, la matrice symétrique $\mathsf{A}_{n,m,p} = (a_{i,j})_{0 \leq i,j \leq n-1}$ de taille $n \times n$, telle que

$$\forall (i,j) \in [\![0, n-1]\!]^2.\ a_{i,j} = \begin{cases} 1 & \text{si } i \leq j \text{ et } (u_{n^2+89m+71p+ni+j} \bmod p) < m; \\ 0 & \text{si } i \leq j \text{ et } (u_{n^2+89m+71p+ni+j} \bmod p) \geq m; \\ a_{j,i} & \text{si } i > j. \end{cases}$$

Question 7 *Écrire un programme qui calcule la matrice $\mathsf{A}_{n,m,p}$. Pour les valeurs de n, m, p suivantes, si $(a_{i,j})_{i,j}$ désignent les coefficients de $\mathsf{A}_{n,m,p}$, donner la somme*

$$\left(\sum_{0 \leq i,j \leq n-1} (ni + j) \times a_{i,j} \right) \bmod 10\,000.$$

a) *n=3, m=1, p=2*

b) *n=20, m=3, p=4*

c) *n=100, m=3, p=4*

d) *n=1 000, m=1, p=1 000*

```c
/* remplit la table Un -- à appeler préalablement dans main() */
void precalcul_un(int u0) {
  Un[0] = u0;
  for (int i = 1; i < max_n; i++) {
    Un[i] = (569 * Un[i-1]) % 2424259;
    assert(569 * Un[i-1] > -1);
  }
  return;
}

/* renvoie u_n */
int un(int n) {
  assert(n > -1 && n < max_n);
  return Un[n];
}

/* calcule A_n,m,p */
int** anmp(int n, int m, int p) {
  int** a = alloue_matrice(n);

  /* remplit d'abord les i <= j */
  for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
      if ((un(n*n + 89*m + 71*p + n*i + j) % p) < m) {
        a[i][j] = 1;
      } else {
        a[i][j] = 0;
      }
    }
  }
```

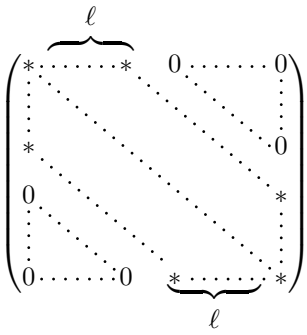
```c
/* puis les i > j */
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
      a[i][j] = a[j][i];
    }
  }

  return a;
}


/* calcul de la somme */
int q7(int n, int m, int p) {
  int** a = anmp(n,m,p);
  int s = 0;
  for (int i=0; i < n; i++) {
    for (int j=0; j < n; j++) {
      s = s + ((n*i + j) * a[i][j]);
    }
  }
  free_matrice(n, a);
  return s % 10000;
}
```

3.2 Largeur de bande

Soit $\mathsf{M} = (m_{i,j})_{0 \leq i,j \leq n-1}$ une matrice carrée de taille $n \times n$ pour un entier $n \geq 1$. On appelle *largeur de bande* de M le plus petit entier ℓ tel que

$$\forall i, j \in [\![0, n-1]\!].\ |i-j| > \ell \implies m_{i,j} = 0.$$

Une matrice de largeur de bande ℓ a donc la forme suivante :



Les seuls coefficients potentiellement non nuls de la matrice se trouvent dans la bande délimitée par $*$, de largeur ℓ de part et d'autre de la diagonale. En particulier, les matrices diagonales sont celles de largeur de bande nulle.

Question 8 *Écrire un programme qui calcule la largeur de bande d'une matrice carrée. Donner la largeur de bande des matrices suivantes.*

a) $\mathsf{A}_{10,1,10}$ ***b)*** $\mathsf{A}_{100,1,100}$ ***c)*** $\mathsf{A}_{200,1,1\,000}$

```c
int largeurbande(int n, int** a) {
  int l = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      if (a[i][j] != 0 && abs(i-j) > l) {
        l = abs(i-j);
      }
    }
  }
  return l;
}


int q8(int n, int m, int p) {
  int** a = anmp(n,m,p);
  int l = largeurbande(n, a);
  free_matrice(n, a);
  return l;
}
```

Un intérêt de manipuler des matrices avec une faible largeur de bande est la possibilité de les stocker plus efficacement, de façon à occuper moins d'espace. En effet, puisque l'on sait que les coefficients en dehors de la bande sont nuls, il suffit de stocker ceux de la bande.

Question à développer pendant l'oral 7

- *Décrire comment ce stockage optimisé d'une matrice pourrait être réalisé en C.*
- *Évaluer l'espace occupé par une matrice stockée sous cette forme, en fonction notamment de sa largeur de bande.*
- *Quelle serait la complexité en temps de l'opération consistant à mettre une matrice stockée comme un tableau $n \times n$ sous cette forme ? Et celle de l'accès à un élément de la matrice ?*

Il n'est pas demandé de réellement programmer cette façon de stocker les matrices.

Dans une matrice M de taille $n \times n$ et de largeur de bande ℓ, les seules valeurs non nulles sont situées dans une bande de largeur ℓ de part et d'autre de la diagonale. Au lieu de stocker toute la matrice, il suffit donc de stocker cette bande. En la tournant de $\frac{\pi}{4}$ dans le sens direct, on la voit comme une table t de $2\ell+1$ lignes (la diagonale, ℓ lignes au dessus, et ℓ en dessous) et n colonnes. Par exemple avec $n = 6$ et $\ell = 1$:

$$\begin{pmatrix} a & b & 0 & 0 & 0 & 0 \\ c & a & b & 0 & 0 & 0 \\ 0 & c & a & b & 0 & 0 \\ 0 & 0 & c & a & b & 0 \\ 0 & 0 & 0 & c & a & b \\ 0 & 0 & 0 & 0 & c & a \end{pmatrix} \longrightarrow \begin{pmatrix} b & b & b & b & b & 0 \\ a & a & a & a & a & a \\ 0 & c & c & c & c & c \end{pmatrix}$$

De façon générale, la table t est telle que lorsque $|i-j| \leq d$:

$$\mathsf{M}[i][j] = t[i-j+d][i]$$

et les cases non utilisées de t sont laissées à 0.

La table t occupe un espace $\mathcal{O}(nd)$, au lieu de $\mathcal{O}(n^2)$ pour la matrice originale. Construire t se fait en un parcours de t, et donc en temps $\mathcal{O}(nd)$ également. Accéder à un élément de M se fait en temps constant.

3.3 Réduction de la largeur de bande

Étant donné une matrice, on peut tenter de réordonner ses lignes et ses colonnes de façon à réduire sa largeur de bande. Cela laisse les solutions du système linéaire associé inchangées (à l'ordre des inconnues près), tout en permettant d'optimiser plus le stockage de la matrice.

Dans le cas d'une matrice symétrique, une méthode qui donne souvent d'assez bons résultats consiste à utiliser pour cela une variante du parcours en largeur.

Soit $\mathsf{M} = (m_{i,j})_{0 \leq i,j \leq n-1}$ une matrice symétrique de taille $n \times n$. On voit M comme la matrice d'adjacence d'un graphe $\mathcal{G}$ non orienté : son ensemble de sommets est $\mathcal{S} = [\![0, n-1]\!]$, et une arête relie i et j lorsque $m_{i,j} = 1$. On autorise une arête à relier un sommet à lui-même. On rappelle que le degré $\deg(i)$ d'un sommet i est le nombre d'arêtes qui y sont reliées. On considère sur $\mathcal{S}$ l'ordre suivant : $i \prec j$ si $\deg(i) < \deg(j)$, ou si $\deg(i) = \deg(j) \wedge i < j$.
L'algorithme consiste, intuitivement, à effectuer un parcours en largeur de $\mathcal{G}$, dans lequel à chaque fois que l'on a le choix de l'ordre dans lequel traiter plusieurs sommets, on les considère suivant l'ordre $\prec$ croissant.

```
F := file vide
Vus := ∅
tant que ∃s ∈ S \ Vus
    s := sommet de S \ Vus minimal pour ≺
    ajouter s à Vus
    enfiler s dans F
    tant que F ≠ file vide
        s_0 := défiler F
        V := voisins de s_0 qui ne sont pas dans Vus
        pour chaque v ∈ V dans l'ordre croissant pour ≺
            ajouter v à Vus
            enfiler v dans F
renvoyer l'ordre d'ajout des sommets à Vus
```

Algorithme 1 : Parcours en largeur

À l'issue de l'algorithme, chaque sommet a été ajouté à l'ensemble Vus exactement une fois. L'ordre dans lequel les sommets ont été ajoutés à Vus fournit une permutation σ : $\sigma(0)$ est le sommet ajouté en premier, $\sigma(1)$ le second, *etc.*
On renumérote les sommets de $\mathcal{G}$ suivant σ pour obtenir un graphe $\mathcal{G}'$: ses sommets sont toujours $\mathcal{S}$, et une arête relie maintenant i et j lorsque $\sigma(i)$ et $\sigma(j)$ étaient reliés dans $\mathcal{G}$.

On note $\mathsf{R}(\mathsf{M})$ la matrice d'adjacence de $\mathcal{G}'$: si $(r_{i,j})_{0 \leq i,j \leq n-1}$ désignent ses coefficients, $r_{i,j}$ vaut 1 lorsque i et j sont reliés par une arête dans $\mathcal{G}'$, et 0 sinon.
Cette matrice $\mathsf{R}(\mathsf{M})$ est en fait la matrice M à laquelle on a appliqué la permutation σ^{-1} à la fois sur les lignes et les colonnes, et a souvent en pratique une plus faible largeur de bande que M.

Question 9 *Écrire un programme qui calcule* $\mathsf{R}(\mathsf{M})$ *à partir de* M. *Pour les matrices* M *suivantes, si l'on note* $\mathsf{R}(\mathsf{M}) = (r_{i,j})_{0 \leq i,j \leq n-1}$, *donner la somme*

$$\left(\sum_{0 \leq i,j \leq n-1} (ni + j) \times r_{i,j} \right) \bmod 10\,000.$$

a) $\mathsf{A}_{10,1,10}$ ***b)*** $\mathsf{A}_{100,1,100}$ ***c)*** $\mathsf{A}_{1\,000,1,1\,000}$ ***d)*** $\mathsf{A}_{1\,000,1,10}$

Indication. Le fichier `base.c` fourni contient une fonction `trie_tableau`, dont le comportement est expliqué en commentaire, qui peut être utile pour cette question.

```c
/* calcule la table des degrés de tous les sommets de a */
int* degres(int n, int** a) {
  int* deg = malloc(n * sizeof(int));
  if (!deg) { erreur("degres malloc deg"); }
  for (int i = 0; i < n; i++) {
    deg[i] = 0;
    for (int j = 0; j < n; j++) {
      if (a[i][j] != 0) {
        deg[i] = deg[i] + 1;
      }
    }
  }
  return deg;
}



/* parcours en largeur de a, renvoie l'ordre de parcours des sommets */
int* parcours(int n, int** a) {

  /* stockage des sommets dans un tableau */
  int* sommets = malloc(n * sizeof(int));
  if (!sommets) { erreur("parcours malloc sommets"); }
  for (int i = 0; i < n; i++) {
    sommets[i] = i;
  }

  /* calcul des degrés */
  int* deg = degres(n, a);

  /* tri par degré croissant */
  trie_tableau(sommets, n, deg);

  /* file implémentée par un tableau et deux indices */
  int* f = malloc(n * sizeof(int)); // on n'aura que n entiers dans la file
  int f_in = 0; // indice où écrire pour enfiler
  int f_out = 0; // indice où écrire pour défiler. f_in == f_out ssi f est vide

  /* tableau pour stocker l'ordre de visite -- stocke vus en même temps */
  int* ordre = malloc(n * sizeof(int));
  if (!ordre) { erreur("parcours malloc ordre"); }
  for (int i = 0; i < n; i++) {
    ordre[i] = -1; // -1 si i pas encore vu
  }
```

```
int o = 0; // ordre du sommet suivant

  for (int si = 0; si < n; si++) {
    int s = sommets[si]; // on parcourt les sommets par degré croissant
    if (ordre[s] == -1) { // on ne fait rien si s déjà vu
      /* à ce stade s est le plus petit sommet non déjà vu */
      ordre[s] = o; // s vu
      o++;
      f[f_in] = s; // enfile s
      f_in++;

      while (f_out < f_in) {
        int s0 = f[f_out]; // defile s0
        f_out++;

        /* calcul des voisins non vus (par degré croissant) */
        for (int j = 0; j < n; j++) {
          int v = sommets[j];
          if ((a[s0][v] != 0) && (ordre[v] == -1)) {
            ordre[v] = o; // v vu
            o++;
            f[f_in] = v; // enfile v
            f_in++;
          }
        }
      }
    }
  }

  assert(o == n); // on doit avoir vu tout le monde
  assert(f_in == f_out); // f doit être vide

  free(f);
  free(deg);
  free(sommets);
  return ordre;
}


/* renumérote les sommets de a (n * n) suivant ordre (ordre = sigma^-1)*/
int** renumerote(int n, int** a, int* ordre) {
  int** ra = alloue_matrice(n);
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      ra[ordre[i]][ordre[j]] = a[i][j];
    }
  }
  return ra;
}
```

```c
int q9(int n, int m, int p) {
  int** a = anmp(n, m, p);
  int* ordre = parcours(n, a);
  int** ra = renumerote(n, a, ordre);
  free(ordre);
  free_matrice(n, a);
  int s = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      s = (s + ((n*i + j) * ra[i][j])) % 10000;
    }
  }
  free_matrice(n, ra);
  return (s % 10000);
}
```

Question à développer pendant l'oral 8 *Expliquer le fonctionnement de votre programme. Évaluer sa complexité en temps.*

Le programme proposé commence par calculer la table `deg` contenant le degré de chaque sommet. Ceci est fait en parcourant tout la matrice pour compter chaque arête, et donc en temps $\mathcal{O}(n^2)$.
On fabrique ensuite la table des sommets, que l'on trie par degré croissant, en temps $\mathcal{O}(n\log(n))$.
On effectue ensuite le parcours en largeur tel qu'il est décrit dans le sujet. La file, qui contient à chaque instant au plus n éléments, est implémentée par un tableau de taille n, ainsi que deux indices indiquant le début et la fin de la file. On parcourt ensuite les sommets dans l'ordre croissant. Dans le pire des cas, chaque sommet devra être considéré : $\mathcal{O}(n)$ tours de boucle. Pour chacun, on calcule ses voisins, que l'on obtient par degré croissant grâce à la table triée précédemment. Ceci est fait en parcourant toute la ligne associée au sommet dans la matrice : $\mathcal{O}(n)$ opérations. En tout, le parcours a donc une complexité en $\mathcal{O}(n^2)$.
Enfin, une fois l'ordre de visite par le parcours calculé, on renumérote les lignes et colonnes de la matrice, en $\mathcal{O}(n^2)$ également.

Tout l'algorithme de réduction de la largeur de bande est donc en $\mathcal{O}(n^2)$.

4 Éclairage sous contrainte

Cette partie doit être traitée en OCaml.

Cette partie est consacrée à une variante du puzzle, dont l'objectif reste le même, à savoir trouver sur quelles cases appuyer pour reproduire une grille cible donnée. On impose une contrainte supplémentaire : il est maintenant obligatoire de sélectionner exactement une case sur chaque ligne de la grille. Il ne s'agit pas d'un cas particulier du puzzle de départ : il se peut que pour une grille donnée, le puzzle non contraint ait une solution, sans pour autant que le puzzle contraint en ait une.

On s'intéressera dans cette partie à déterminer l'existence d'une solution, plutôt qu'à la recherche d'une solution particulière.
Plus précisément, étant donné une grille cible G de taille $n \times n$, on souhaite calculer pour chaque entier $d \in [\![0, n^2]\!]$ le booléen $\mathsf{E}_\mathsf{G}(d)$, qui est vrai lorsqu'il existe un ensemble de cases qui respecte la contrainte (une case par ligne), et dont l'image est à distance d de G :

$$\mathsf{E}_\mathsf{G}(d) \iff \exists \mathsf{C} \text{ respectant la contrainte. } \mathsf{d}(\mathsf{G}, \mathcal{I}(\mathsf{C})) = d.$$

4.1 Recherche exhaustive

On propose, pour commencer, de résoudre cette variante du problème par une approche exhaustive. On souhaite donc tester un par un tous les choix possibles d'une case (et une seule) par ligne, pour déterminer quelles distances d peuvent être atteintes.

Question 10 *Écrire un programme qui effectue cette recherche exhaustive. Donner, pour les entrées suivantes, la somme* (mod 10 000) *des valeurs de* d *telles que* $\mathsf{E}_\mathsf{G}(d)$ *est vrai :*

$$\left(\sum_{d \in [\![0, n^2]\!] \text{ tel que } \mathsf{E}_\mathsf{G}(d)} d \right) \bmod 10\,000.$$

a) $\mathsf{G}_{3,1,2}$ ***b)*** $\mathsf{G}_{5,3,7}$ ***c)*** $\mathsf{G}_{6,3,7}$

```
let forcebrute_une (g:grille) : bool array =
  let n = taille g in
  let h = vide n in (* grille de travail *)
  let eg = Array.make (n * n + 1) false in

  (* écrit dans eg.(d) "true" si il existe un choix pour les lignes <= i qui donnent
     une distance de d quand on choisit une case <= j sur la ligne i *)
  let rec aux i j =
    if i < 0 then
      let d = distance h g in
      eg.(d) <- true;
    else
      if j >= 0 then
        (appuie h (i,j);
         aux (i-1) (n-1);
         appuie h (i,j);
         aux i (j-1))
  in
  aux (n-1) (n-1);
  eg

let q10 (n:int) (m:int) (p:int) : int =
  let g = gnmp n m p in
  let eg = forcebrute_une g in
  let s = ref 0 in
  Array.iteri (fun d nn -> if nn then s := !s + d) eg;
  (!s mod 10000)
```

Question à développer pendant l'oral 9 *Évaluer la complexité en temps de votre programme.*

Le programme proposé parcourt tous les choix de cases satisfaisant la contrainte : on choisit librement une case parmi les n sur chacune des n lignes, et il y a donc n^n choix possibles. Pour chaque choix de cases, on calcule la distance résultante, ce qui a une complexité en $\mathcal{O}(n^2)$. Par conséquent, le programme a une complexité en temps $\mathcal{O}(n^{n+2})$.

4.2 Calcul efficace

La recherche exhaustive n'est pas très efficace, et ne permet pas de calculer le résultat souhaité pour de grandes tailles de grille.

Question 11 *Écrire un programme qui calcule efficacement la fonction* E. *Donner pour les entrées suivantes, comme précédemment, la somme* $\bmod 10\,000$ *des valeurs de* d *telles que* $\mathsf{E}_{\mathsf{G}}(d)$ *est vrai :*

$$\left(\sum_{d \in [\![0, n^2]\!] \text{ tel que } \mathsf{E}_{\mathsf{G}}(d)} d \right) \bmod 10\,000.$$

a) $\mathsf{G}_{12,3,7}$ ***b)*** $\mathsf{G}_{16,5,7}$ ***c)*** $\mathsf{G}_{20,3,7}$

Indication. On pourra utiliser des techniques de programmation dynamique.

```
let dyn (g:grille) : bool array =
  let n = taille g in
  let h = Array.make_matrix n n (Array.make_matrix 0 0 false) in
  for i=0 to (n-1) do
    for j=0 to (n-1) do
      h.(i).(j) <- Array.make_matrix n (n*n + 1) false;
    done;
  done;

  let dligne = Array.make_matrix n n 0 in
  for i = 0 to n - 1 do
    let d = ref 0 in (* distance sans rien appuyer *)
    for l = 0 to n - 1 do
      if est_allume g (i, l) then
        incr d;
    done;
    for j = 0 to n - 1 do (* pour chaque j, on corrige au voisinage de j *)
      dligne.(i).(j) <- !d;
      List.iter
        (fun l -> (* si (i, l) est censée être allumée : -1, sinon +1 *)
          if est_allume g (i, l) then
            dligne.(i).(j) <- dligne.(i).(j) - 1
          else
            dligne.(i).(j) <- dligne.(i).(j) + 1)
        (List.filter (fun l -> l >= 0 && l < n) [j - 1; j; j + 1]);
    done;
  done;
  let dligne i j = dligne.(i).(j) in

  let dint_haut i j k =
    if xor (k = j - 1 || k = j || k = j + 1) (est_allume g (i, k)) then
      -1
    else
      1
  in

  let dint_bas i j k l =
    if xor (xor (j = k - 1 || j = k || j = k + 1) (j = l)) (est_allume g (i - 1, j))
    ↪  then
      -1
    else
      1
  in
```

```
  for j=0 to (n-1) do
      for k=0 to (n-1) do
        let dd = (dligne 1 j) + (dligne 0 k) + (dint_haut 1 j k) + (dint_haut 0 k j)
        ↪  in
        h.(1).(j).(k).(dd) <- true;
      done;
    done;

    for i=2 to (n-1) do
      for j=0 to (n-1) do
        for k=0 to (n-1) do
          for d=0 to (n*n) do
            let dd = d - (dligne i j) - (dint_haut i j k) in
            let l = ref 0 in
            while (!l < n && not h.(i).(j).(k).(d)) do
              let ddd = dd - (dint_bas i j k !l) in
              if ddd >= 0 && ddd <= n*n && h.(i-1).(k).(!l).(ddd) then
                h.(i).(j).(k).(d) <- true;
              incr l;
            done;
          done;
        done;
      done;
    done;

    let eg = Array.make (n * n + 1) false in
    for d=0 to n*n do
      let j = ref 0 in
      while (!j < n && not eg.(d)) do
        let k = ref 0 in
        while (!k < n && not eg.(d)) do
          if h.(n-1).(!j).(!k).(d) then
            eg.(d) <- true;
          incr k;
        done;
        incr j;
      done;
    done;
    eg

let q11 (n:int) (m:int) (p:int) : int =
    let g = gnmp n m p in
    let eg = dyn g in
    let s = ref 0 in
    Array.iteri (fun d nn -> if nn then s := !s + d) eg;
    (!s mod 10000)
```

Question à développer pendant l'oral 10 *Décrire le fonctionnement de votre programme. Évaluer sa complexité en temps et en espace.*

Suivant l'indication, on utilise un algorithme dynamique. L'idée naturelle serait de considérer le problème ligne par ligne, en se disant que, pour atteindre une distance d avec seulement les lignes 0 à i, on examine chacune des n possibilités pour la case sur la ligne i, et on détermine dans chaque cas quelle serait la distance d' restant à atteindre avec seulement les lignes 0 à $i-1$. Si l'une de ces d' est atteignable, alors d l'est aussi. On pourrait ainsi calculer successivement les distances atteignables avec la ligne 0, puis 0 à 1, 0 à 2, …, 0 à n, ce qui résoudrait la question.

Il y a cependant une subtilité : savoir que l'on a décidé de choisir la case j sur la ligne i ne suffit pas à déterminer la distance restant à atteindre avec les lignes $< i$. En effet, cette distance diffère selon le choix k qui sera fait ligne $i-1$. Appuyer sur $(i-1,k)$ modifiera aussi (i,k), ce qui va augmenter ou diminuer la distance selon 1) l'état de la case (i,k) dans la cible, et 2) que (i,k) a été modifiée lors de l'appui sur (i,j), ce qui dépend de $|k-j|$.
Par conséquent, notre algorithme dynamique exploitera une relation de récurrence qui demandera de considérer tous les cas possibles pour la dernière et l'avant- dernière ligne.
On construira une table h de n^5 booléens, à 4 dimensions : $h[i][j][k][d]$ est vrai lorsqu'il existe un choix de cases sur les lignes 0 à i, contenant les cases (i,j) et $(i-1,k)$, qui atteint la distance d lorsque l'on ne prend en compte que les lignes 0 à i.
On a alors la relation suivante : pour tous i,j,k,d, avec $i>1$,

$$h[i][j][k][d] = \bigvee_{0\leq l\leq n-1} h[i-1][k][l][d-d'],$$

où $d' = \mathsf{d}^{i,j}_{\mathsf{ligne}} + \mathsf{d}^{i,j,k}_{\uparrow} + \mathsf{d}^{i,j,k,l}_{\downarrow}$ est la variation de distance causée par l'appui sur (i,j) :

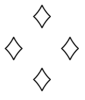
- $\mathsf{d}^{i,j}_{\mathsf{ligne}}$ est la distance entre la ligne i de G et une ligne sur laquelle on a uniquement appuyé sur la j-ième case.
- $\mathsf{d}^{i,j,k}_{\uparrow}$ est la distance introduite sur la ligne i par l'appui sur $(i-1,k)$, lorsque l'on choisit la case (i,j). Elle vaut 1 ou -1, selon que k est assez proche de j ou non, et que la case (i,k) de G est allumée ou non.
- $\mathsf{d}^{i,j,k,l}_{\downarrow}$ est la distance introduite sur la ligne $i-1$ par l'appui sur (i,j), sachant que l'on choisit les cases $(i-1,k)$ et $(i-2,l)$. Elle vaut 1 ou -1, selon la proximité de j, k, l et l'état de la case $(i-1,j)$ dans G.

On initialise la ligne $h[1]$ (la ligne $h[0]$ n'a pas de sens, et n'est pas utilisée), en utilisant la relation :

$$h[1][j][k][d] \Leftrightarrow d = \mathsf{d}^{1,j}_{\mathsf{ligne}} + \mathsf{d}^{0,k}_{\mathsf{ligne}} + \mathsf{d}^{1,j,k}_{\uparrow} + \mathsf{d}^{0,k,j}_{\uparrow}.$$

On remplit ensuite successivement toutes les lignes de h, par valeur i croissante. Une fois h remplie, il ne reste pour calculer $\mathsf{E}_{\mathsf{G}}(d)$ pour un certain d qu'à prendre la disjonction de tous les $h[n-1][j][k][d]$, pour toutes les valeurs de j, k.

La distance $\mathsf{d}^{i,j}_{\mathsf{ligne}}$ peut être précalculée pour tous les (i,j) en $\mathcal{O}(n^2)$, $\mathsf{d}^{i,j,k}_{\uparrow}$ et $\mathsf{d}^{i,j,k,l}_{\downarrow}$ se calculent en temps constant. Lors du remplissage de h, on calcule la disjonction de n valeurs précédemment calculées, pour chacune des $\mathcal{O}(n^5)$ cases de h. Le calcul final de $\mathsf{E}_{\mathsf{G}}(d)$ est enfin en $\mathcal{O}(n^2)$ opérations pour chaque d. On obtient ainsi au total un algorithme de complexité temporelle $\mathcal{O}(n^6)$.
Quant à la complexité en espace : la table h la fait passer à $\mathcal{O}(n^5)$. On pourrait, comme souvent avec les algorithmes dynamiques, ne conserver que la dernière ligne de h calculée, pour obtenir une complexité en espace $\mathcal{O}(n^4)$.



Fiche réponse type : Lumière

$\widetilde{u_0}$: 1 571

Question 1

a) 3 899

b) 617

c) 9 837

d) 394

Question 2

a) 5

b) 698

c) 8 745

Question 3

a) 1

b) 50

c) 4 890

Question 4

a) 10

b) 36

c) 39

Question 5

a) 12

b) 1 447

c) 2 630

Question 6

a) 82

b) 4 347

c) 8 387

d) 7 660

Question 7

a) 20

b) 5 016

c) 79

d) 555

Question 8

a) 9

b) 77

c) 163

Question 9

a) 968

b) 6 693

c) 6 905

d) 6 584

Question 10

a) 26

b) 225

c) 425

Question 11

a) 4 623

b) 5 939

c) 4 991

Réécriture et partage

Épreuve pratique d'algorithmique et de programmation
Concours commun des Écoles normales supérieures

Durée de l'épreuve : 3 heures 30 minutes

Juin/Juillet 2024

ATTENTION !

N'oubliez en aucun cas de recopier votre u_0
à l'emplacement prévu sur votre fiche réponse

Important.

Il vous a été donné un numéro u_0 qui servira d'entrée à vos programmes. Les réponses attendues sont généralement courtes et doivent être données sur la fiche réponse fournie à la fin du sujet. À la fin du sujet, vous trouverez en fait deux fiches réponses. La première est un exemple des réponses attendues pour un $\widetilde{u_0}$ particulier (précisé sur cette même fiche et que nous notons avec un tilde pour éviter toute confusion !). Cette fiche est destinée à vous aider à vérifier le résultat de vos programmes en les testant avec $\widetilde{u_0}$ au lieu de u_0. Vous indiquerez vos réponses (correspondant à votre u_0) sur la seconde et vous la remettrez à l'examinateur à la fin de l'épreuve.

En ce qui concerne la partie orale de l'examen, lorsque la description d'un algorithme est demandée, vous devez présenter son fonctionnement de façon schématique, courte et précise. Vous ne devez en aucun cas recopier le code de vos procédures !

Quand on demande la complexité en temps ou en mémoire d'un algorithme en fonction d'un paramètre n, on demande l'ordre de grandeur en fonction du paramètre, par exemple : $O(n^2)$, $O(n \log n)$,...

Il est recommandé de commencer par lancer vos programmes sur de petites valeurs des paramètres et de ***tester vos programmes sur des petits exemples que vous aurez résolus préalablement à la main ou bien à l'aide de la fiche réponse type fournie en annexe***. Enfin, il est recommandé de lire l'intégralité du sujet avant de commencer afin d'effectuer les bons choix de structures de données dès le début.

Les Parties 1 et 3 (OCaml) sont indépendantes de la Partie 2 (C).

Le fichier `base.ml` correspond aux Parties 1 et 3, et le fichier `base.c` à la Partie 2. Le dossier `data/` contient les jeux de tests.

Il est recommandé de garder une sauvegarde de tout les fichiers fournis, au cas où ceux-ci seraient modifiés par erreur.

Il est précisé dans chaque partie le langage d'implémentation à utiliser. Il est demandé de nous fournir sur votre clef USB vos fichiers OCaml et C. Ces consignes doivent impérativement être suivies.

Préliminaires

Jeux de tests et entrées Plusieurs jeux de tests sont disponibles dans le répertoire `data/`. Chacun se trouve dans un répertoire `data/U0/`, où `U0` est la valeur u_0. Ces jeux de tests sont utilisés dans les Parties 1 et 3. Des fonctions en OCaml `read_terms`, `read_rule_terms` et `read_rules` servant à la lecture depuis les fichiers sont fournies dans le fichier `base.ml`.

Résultats et évaluation Votre évaluation portera sur les fichiers dont l'`U0` correspond à votre u_0. Les autres fichiers peuvent être utilisés pour tester vos programmes.

Notations On rappelle que pour deux entiers naturels a et b, avec $b \neq 0$, $(a \mod b)$ désigne le reste de la division euclidienne de a par b, c'est-à-dire l'unique entier r avec $0 \leq r < b$ tel qu'il existe un $k \in \mathbb{N}$ vérifiant $a = k \times b + r$.

1 Partie 1 (OCaml) : Réécriture de termes

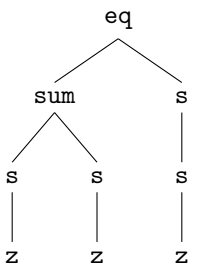
Vous devez utiliser OCaml tout au long de cette partie.

Code fourni Le fichier `base.ml` contient les éléments de code suivants :

- Un module `Tag` permettant de créer des tags de façon abstraites.
- Les déclarations des types `term`, `rule_term` et `rule`.
- Les fonctions `read_terms`, `read_rule_terms` et `read_rules` permettant de lire les fichiers de données dans `data/U0`.
- Un module `HashCons` utilisé dans la Partie 3.

Les modules, types et fonctions fournis sont décrits plus bas lorsque ceux-ci sont utilisés.

Termes Un terme est un arbre non-vide dont les nœuds sont étiquetés par des symboles (représentés par des chaînes de caractères). Un nœud interne est un nœud avec au moins un enfant, et une feuille un nœud sans enfant. Par exemple, le terme t_0 suivant :



a trois feuilles, toutes étiquetées par `z`, et six nœuds internes étiquetés par `s`, `sum` et `eq`.
Un terme peut aussi être représenté par un mot, dans lequel chaque nœud du terme est suivi par la liste de ses enfants, entre parenthèses et séparés par des virgules. Ainsi, une feuille étiquetée par `c` (n'ayant donc pas d'enfant) est représentée par `c()`. On s'autorisera à enlever les parenthèses pour les symboles sans enfant, et à écrire `c` à la place de `c()`. Par exemple, le terme t_0 peut être représenté par :

$$\texttt{eq}(\texttt{sum}(\texttt{s}(\texttt{z}), \texttt{s}(\texttt{z})), \texttt{s}(\texttt{s}(\texttt{z}))) \tag{1}$$

Chaque nœud d'un terme s définit un sous-terme de s : par exemple, `z` et $\texttt{sum}(\texttt{s}(\texttt{z}), \texttt{s}(\texttt{z}))$ sont des sous-termes de t_0. Notez qu'un terme s est un sous-terme de lui-même.

On utilisera le type OCaml suivant pour représenter les termes :

```
type term = Tfun of Tag.t * string * term list
```

où la construction OCaml `Tfun (tag, symb, [t1 ; ...; tn])` représente le terme `symb(t1, ..., tn)`, dont la racine est étiquetée par `symb` et possède n fils `t1`, ..., `tn`. Le tag `tag` ne sera pas utilisé avant la Section 3. Pour l'instant, on utilisera le tag par défaut `Tag.default` à chaque fois qu'un tag est nécessaire. En OCaml, le terme t_0 est représenté par :

```
let d = Tag.default in
let one = Tfun (d, "s", [Tfun (d, "z", [])]) in   (*   s(z)   *)
let two = Tfun (d, "s", [one])                in   (* s(s(z)) *)
Tfun (d, "eq", [Tfun (d, "sum", [one; one]); two])
```

Question 1 *Implémenter la fonction de signature* `val hash : term -> int`, *telle que pour tout terme* $\texttt{f}(t_1, \ldots, t_n)$:

$$\texttt{hash}\big(\texttt{f}(t_1, \ldots, t_n)\big) = \Big(\big(\texttt{Hashtbl.hash f}\big) + \sum_{i=1}^{n} i \times \texttt{hash}(t_i)\Big) \mod 104\,729$$

où la fonction `Hashtbl.hash` *est fournie par la bibliothèque standard OCaml.*
Notons `terms = read_terms "data/U0/q1.txt"` *(en remplaçant* `U0` *par votre* u_0*).*
Calculer `hash (List.nth terms k)` *pour les valeurs de* `k` *suivantes :*

a) $k = 0$ ***b)*** $k = 1$ ***c)*** $k = 2$ ***d)*** $k = 3$

```
let rec hash (t : term) : int =
  let Tfun (_, s, args) = t in
  let h = ref (Hashtbl.hash s mod n) in
  List.iteri (fun i arg ->
      h := (!h + (( (i + 1) * hash arg) mod n)) mod n
    ) args;
  !h
```

1.1 Substitution et *matching*

Un terme de règle est un terme dont certaines feuilles sont étiquetées par des variables, prises dans l'ensemble de variables $\mathcal{V} = \{\texttt{x}_0, \texttt{x}_1, \ldots, \texttt{x}_9\}$. Dans ce sujet, nous n'aurons jamais besoin de plus de 10 variables. Par exemple, $\texttt{sum}(\texttt{x}_0, \texttt{s}(\texttt{x}_1))$ et $\texttt{eq}(\texttt{x}_0, \texttt{x}_0)$ sont deux termes de règle. En OCaml, on représentera une variable $\texttt{x}_i$ par l'entier i, et les termes de règle par le type :

```
type rule_term =
  | Rvar of int (** on représente chaque variable par un entier *)
  | Rfun of string * rule_term list
```

Ainsi, $\texttt{sum}(\texttt{x}_0, \texttt{s}(\texttt{x}_1))$ est représenté par `Rfun("sum", [Rvar 0; Rfun ("s", [Rvar 1])])`.
Une substitution est une fonction associant des termes à des variables. Par exemple,

$$\sigma_0 = \{\texttt{x}_0 \mapsto \texttt{s}(\texttt{z});\ \ \texttt{x}_1 \mapsto \texttt{z};\ \ \texttt{x}_2 \mapsto \texttt{h}(\texttt{b}, \texttt{s}(\texttt{z}))\} \tag{2}$$

associe $\texttt{s}(\texttt{z})$ à $\texttt{x}_0$, $\texttt{z}$ à $\texttt{x}_1$, $\texttt{h}(\texttt{b}, \texttt{s}(\texttt{z}))$ à $\texttt{x}_2$, et rien aux autres variables.
Naturellement, on peut appliquer une substitution σ à un terme de règle s pour obtenir un terme $\sigma(s)$, en remplaçant chaque variable $\texttt{x}_i$ apparaissant dans s par $\sigma(\texttt{x}_i)$. Par exemple :

$$\sigma_0(\texttt{sum}(\texttt{x}_0, \texttt{s}(\texttt{x}_1))) = \texttt{sum}(\texttt{s}(\texttt{z}), \texttt{s}(\texttt{z}))$$

Si une variable $\texttt{x}_i$ apparaissant dans le terme de règle n'est associée à aucun terme par σ, on substituera celle-ci par le terme `missing()`.

Question 2 *Choisir un type de données, que l'on appellera* `subst`*, pour représenter les substitutions. Implémenter la fonction appliquant une substitution à un terme de règle, de signature* `val subst : subst -> rule_term -> term`.

Notons `rule_terms = read_rule_terms "data/U0/q2.txt"`.
Soit `sigma0` *la substitution* σ_0 *donnée dans l'équation* (2) *ci-dessus. Calculer* `hash (subst sigma0 (List.nth rule_terms k))` *pour les valeurs de* `k` *suivantes :*

a) $k = 0$ ***b)*** $k = 1$ ***c)*** $k = 2$ ***d)*** $k = 3$

```
type subst = term option array

let tfun s l = Tfun (Tag.default, s, l)

let get (s : subst) (v : int) : term =
  match s.(v) with Some x -> x | _ -> tfun "missing" []

let rec subst (s : subst) ( t : rule_term) : term =
  match t with
  | Rvar v -> get s v
  | Rfun (f, l) -> tfun f (List.map (subst s) l)
```

Question à développer pendant l'oral 1 *Donner la complexité en temps de votre algorithme.*

Complexité en temps de `subst s t`.
Implémentation des substitutions par des tableaux : $\mathcal{O}(|\texttt{t}|)$
- accès au tableau de la substitution en O(1)
- pas de copie des termes lors de la substitution d'une variables.

Implémentation des substitutions par des listes d'associations : $\mathcal{O}(|\texttt{t}| * V)$
- V : nombres de variables dans `t` (donc $V \leq |\texttt{t}|$)
- accès au tableau de la substitution en $\mathcal{O}(V)$

Soit t un terme et l un terme de règle. On dit que t *matche* l lorsqu'il existe une substitution σ telle que $t = \sigma(l)$. Par exemple :

$$\begin{array}{rcll} \mathtt{sum}(\mathtt{s}(\mathtt{z}), \mathtt{s}(\mathtt{z})) & \textit{matche} & \mathtt{sum}(\mathtt{x}_0, \mathtt{s}(\mathtt{x}_1)) & \text{avec } \sigma = \{\mathtt{x}_0 \mapsto \mathtt{s}(\mathtt{z});\ \mathtt{x}_1 \mapsto \mathtt{z}\} \\ \mathtt{eq}(t, t) & \textit{matche} & \mathtt{eq}(\mathtt{x}_0, \mathtt{x}_0) & \text{avec } \sigma = \{\mathtt{x}_0 \mapsto t\} \\ \mathtt{eq}(t, t') & \textit{ne matche pas} & \mathtt{eq}(\mathtt{x}_0, \mathtt{x}_0) & \text{quand } t \neq t' \end{array}$$

Question 3 *Implémenter une fonction de signature :*

```
val find_match : term -> rule_term -> subst option
```

telle que si `t` *matche* `l`*, alors* `find_match t l` *renvoie* `Some sigma` *pour un* `sigma` *tel que* `t = subst sigma l`*, et sinon* `find_match t l` *renvoie* `None`*.*

Notons `terms = read_terms "data/U0/q3.txt"`*.*
Soit `t` *le k-ième terme de* `terms`*. Si* `find_match t r = Some sigma`*, calculer la valeur* `v` *de* `hash (subst sigma (Rvar 0))) + hash (subst sigma (Rvar 1)))` *et noter* `Some v` *sur la fiche réponse (sinon, noter* `None`*) pour les valeurs de* k *et* `r` *suivantes :*

a) $k = 0$ *et* $r = \mathtt{eq}(x_0, x_0)$

b) $k = 1$ *et* $r = \mathtt{eq}(x_0, x_0)$

c) $k = 2$ *et* $r = \mathtt{F}(\mathtt{f}(x_0), x_1, \mathtt{g}(x_1))$

d) $k = 3$ *et* $r = \mathtt{F}(\mathtt{f}(x_0), x_1, \mathtt{g}(x_1))$

```
let equal t t' = t = t'

let find_match (t1 : term) (t2 : rule_term) : subst option =
  let exception NoMatch in
  let no_match () = raise NoMatch in

  let s = Array.make max_var None in

  let rec doit t1 t2 =
    let Tfun (_,f1,l1) = t1 in
    match t2 with
    | Rfun (f2, l2) ->
      if f1 <> f2 then no_match ();
      List.iter2 doit l1 l2

    | Rvar v ->
      match s.(v) with
      | None   -> s.(v) <- Some t1
      | Some t -> if not (equal t t1) then no_match ()
  in
  try doit t1 t2; Some s with NoMatch -> None
```

On dit qu'un terme de règle `l` est linéaire si aucune variable $\mathtt{x}_i$ n'apparaît plus d'une fois dans `l`. Par exemple, $\mathtt{sum}(x_0, x_1)$ est linéaire mais pas $\mathtt{eq}(x_0, x_0)$.

Question à développer pendant l'oral 2 *Donner la complexité en temps de* `find_match t l` *en fonction de la taille de* `t` *et* `l` *dans les cas suivants :*

— *le terme* `l` *est linéaire.*

— *dans le cas général, sans hypothèse sur* `l`*.*

Complexité de `find_match t1 t2`.
Complexité en temps (`t2` linéaire) : $\mathcal{O}(\min(|\texttt{t1}|, |\texttt{t2}|))$.

— `doit` est appelé au plus une fois par symbole dans `t1` (idem pour `t2`)

— toutes les autres opérations sont en $\mathcal{O}(1)$ car le test d'égalité dans le cas ou une variable est vu deux fois n'a jamais lieu.

Complexité en temps (cas général) : $\mathcal{O}(|\texttt{t1}| \cdot \min(|\texttt{t1}|, |\texttt{t2}|))$

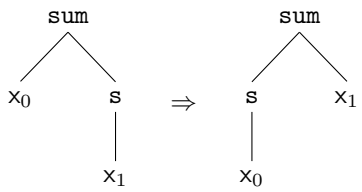
— chaque test d'égalité est en temps $\mathcal{O}(|\texttt{t1}|)$ (car on compare l'égalité de deux sous-termes de `t1`)

— il y a au plus un test d'égalité par noeuds de `t1`. Idem pour `t2`. Donc au plus $\min(|\texttt{t1}|, |\texttt{t2}|)$ tests.

1.2 Système de réécriture

Un système de réécriture $\mathcal{R}$ est un ensemble fini de règles permettant de réécrire dans un terme. Une règle de réécriture R est de la forme $l \Rightarrow r$, où l et r sont des termes de règle. Cette règle signifie que l'on peut réécrire l en r dans **n'importe quel sous-terme** d'un terme. Par exemple, la règle R_0 suivante (représentée à gauche avec des mots, et à droite avec des arbres) :

$$\texttt{sum}(\mathsf{x}_0, \texttt{s}(\mathsf{x}_1)) \Rightarrow \texttt{sum}(\texttt{s}(\mathsf{x}_0), \mathsf{x}_1)$$



(la règle R_0)

utilise deux variables x_0 et x_1. En OCaml, on représentera une règle $l \Rightarrow r$ par le couple (l, r), à l'aide du type suivant :

```
type rule = rule_term * rule_term
```

On écrit $t \to_{\mathsf{R}} t'$ quand le terme t peut se réécrire en t' par la règle R. Par exemple, on a :

$$\texttt{sum}(\texttt{s}(\texttt{z}), \texttt{s}(\texttt{z})) \to_{\mathsf{R}_0} \texttt{sum}(\texttt{s}(\texttt{s}(\texttt{z})), \texttt{z})$$

Une règle de réécriture peut s'appliquer en profondeur, et plusieurs règles peuvent s'appliquer au même terme. Ainsi, en notant R_1 la règle $\texttt{eq}(\mathsf{x}_0, \mathsf{x}_0) \Rightarrow \texttt{true}$, on a (on souligne le sous-terme où la réécriture a lieu) :

$$\begin{aligned} \texttt{pair}(\underline{\texttt{eq}(\texttt{z},\texttt{z})},\ \texttt{eq}(\texttt{z},\texttt{z})) &\to_{\mathsf{R}_1} \texttt{pair}(\underline{\texttt{true}}\ ,\ \texttt{eq}(\texttt{z},\texttt{z})) \\ \texttt{pair}(\texttt{eq}(\texttt{z},\texttt{z}),\ \underline{\texttt{eq}(\texttt{z},\texttt{z})}) &\to_{\mathsf{R}_1} \texttt{pair}(\texttt{eq}(\texttt{z},\texttt{z}),\ \underline{\texttt{true}}\ ) \end{aligned}$$

Plus généralement, détaillons les étapes de la réécriture par une règle $l \Rightarrow r$, dans le cas particulier de l'exemple ci-dessus à gauche :

— On identifie le sous-terme s où l'on applique la réécriture : ici, s est le sous-terme $\mathtt{eq}(\mathtt{z},\mathtt{z})$ de gauche.
— On *matche* l avec le sous-terme s pour obtenir une substitution σ telle que s soit égal à $\sigma(l)$. Ici, on prend $\sigma = \{\mathtt{x}_0 \mapsto \mathtt{z}\}$.
— On remplace le sous-terme s par $\sigma(r)$.

Question 4 *Implémenter la fonction de signature :*

```
val rewrite1 : term -> rule list -> term option
```

telle que `rewrite1 t rules` *réécrive une des règles de* `rules` *dans* `t`*. Si aucune règle de* `rules` *ne s'applique,* `rewrite1` *renverra* `None`*.*
On fera la première réécriture possible sur un sous-terme de `t` *dans une recherche en profondeur, c'est-à-dire que l'on parcourra* `t` *en profondeur jusqu'à atteindre le premier sous-terme auquel l'une des règles s'applique. Si plusieurs règles s'appliquent à ce sous-terme, on utilisera celle qui apparaît en premier dans* `rules` *lors d'un parcours de la liste.*

Notons `terms = read_terms "data/U0/tq4.txt"` *et* `rules = read_rules "data/U0/rq4.txt"`*. Soit* `t` *le k-ième terme de* `terms`*. Si* `rewrite1 t rules = Some t'`*, alors noter* `Some (hash t')` *sur la fiche réponse (sinon, noter* `None`*), pour les valeurs de k suivantes :*

a) $k = 0$ ***b)*** $k = 1$ ***c)*** $k = 2$ ***d)*** $k = 3$

```
let rewrite1 (t : term) (rules : rule list) : term option =
  let found = ref false in

  (* cherche à appliquer une règle dans [t] *)
  let rec doit t =
    if !found then t else
      (* on cherche une règle s'appliquant en tête *)
      match t with
      | Tfun (_, f, l) as t ->
        let t = (* itére sur les règles *)
          List.find_map (fun (rl, rr) ->
              match find_match t rl with
              | None   -> None

              (* [s] tel que [t = subst s rl], on renvoie [subst s rr] *)
              | Some (s : subst) -> Some (subst s rr)
            ) rules
        in
        match t with
        | Some t -> found := true; t   (* match trouvé en position de tête *)

        (* pas de match en position de tête dans [t], on récurse *)
        | None -> tfun f (List.map doit l)
  in
  let t = doit t in
  if !found then Some t else None
```

On généralise la notion de linéarité comme suit : on dit qu'un terme de règle `l` est L-linéaire si L est le plus petit entier tel qu'aucune variable n'apparaisse plus de L fois dans `l`. Par exemple, $\mathtt{f}(\mathtt{x}_0, \mathtt{x}_0, \mathtt{x}_1)$ est 2-linéaire car $\mathtt{x}_0$ apparaît deux fois.

Question à développer pendant l'oral 3 *Donner la complexité en temps et borner la taille de la sortie de* `rewrite1 t rules`*. On exprimera ces deux quantités en fonction, entre autres, des paramètres suivants :*

- *le nombre N de règles dans* `rules`*, et le nombre V de variables différentes dans les règles ;*
- *une borne L sur la L-linéarité des termes de règles apparaissant à droites des règles de* `rules`*, c.-à-d. que pour chaque règle* `l` $\Rightarrow$ `r` *de* `rules`*, le terme* `r` *est L'-linéaire pour $L' \leq L$.*

Analyse de la fonction `rewrite1 t rules`.

- N règles dans `rules` avec au plus V variables différentes.
- chaque règle `(l,r)` de `rules` est L-linéaire à droite, et $|\mathtt{r}| < R$

Complexité en temps : $\mathcal{O}(|\mathtt{t}|^3 \cdot N)$

- pour chaque sous-terme de `t` et règle dans `rules`, donc $|\mathtt{t}| \cdot N$ fois
- on appel `find_match`, en temps $\mathcal{O}(|t|^2)$ (on simplifie la borne précédente).

Borne sur la taille de la sortie : $|\mathtt{t}| + |\mathtt{r}| + V \cdot (L-1) \cdot |\mathtt{t}|$

- on raisonne en regardant ce qu'on ajoute au terme `t` initial
- supposons que l'on réécrive la règle `(l,r)`
- on enlève $|\mathtt{l}|$ noeuds et on en ajoute $|\mathtt{r}|$, donc ajoute au plus $|\mathtt{r}|$ noeuds
- de plus, pour chaque variable `v`, on a au plus $L-1$ copie d'un sous-terme de `t`, donc un ajoute au plus $V \cdot (L-1) \cdot |\mathtt{t}|$ noeuds

Question à développer pendant l'oral 4 *Simplifier les expressions des complexités en temps et des bornes sur la taille de la sortie précédentes dans le cas où l'ensemble des règles* `rules` *est supposé fixé.*

En supposant l'ensemble des règles fixé.
Borne sur la taille de la sortie :

- si $L = 1$, alors on a $|\mathtt{t}| + |\mathtt{r}| = |\mathtt{t}| + K$ pour une certaine constante K.
- si $L > 1$, alors on a $C \cdot |\mathtt{t}|$ pour un certain entier $C > 0$.

Une séquence de réécritures au départ d'un terme t pour un ensemble de règles $\mathcal{R}$ est une suite de termes $t_0, \ldots, t_l$ telle que $t_0 = t$ et $t_0 \to_{\mathcal{R}} t_1 \to_{\mathcal{R}} \ldots \to_{\mathcal{R}} t_l$. La longueur d'une séquence est le nombre de réécritures dans celle-ci : ainsi, la séquence précédente est de longueur l. Un terme t est dit irréductible pour $\mathcal{R}$ si aucune règle de $\mathcal{R}$ ne s'applique à t, c-.à.d. si il n'existe pas de terme t' tel que $t \to_{\mathcal{R}} t'$.

Question 5 *Implémenter la fonction de signature :*

```
val rewriteN : term -> rule list -> term * int
```

telle que `rewriteN t rules` *applique la fonction* `rewrite1` *de façon répéter pour calculer une séquence de réécritures au départ de* `t` *par* `rules` *jusqu'à obtenir un terme* `t'` ***irréductible****.*
La fonction `rewriteN t rules` *renvoie le couple* `(t', l)` *composé du terme finale* `t'` *et de la longueur* `l` *de la séquence de réécritures calculée. Plus précisément, on a* `t'` *irréductible tel que*

$$\texttt{t} = \texttt{t}_0 \to_{\mathcal{R}} \ldots \to_{\mathcal{R}} \texttt{t}_l = \texttt{t'} \quad \textit{avec} \quad (\texttt{rewrite1 t}_i\ \texttt{rules} = \texttt{t}_{i+1}) \quad \textit{pour tout } 0 \leq i < l.$$

On admet que ce processus termine, c.-à.d. que `t'` ***et*** `l` ***sont bien définis sur les entrées données par l'énoncé.***

Notons `terms = read_terms "data/U0/tq5.txt"` *et* `rules = read_rules "data/U0/rq5.txt"`*. Soit* `t` *le k-ième terme de* `terms`*. Calculer l'entier* `snd (rewriteN t rules)` *pour les valeurs de k suivantes :*

a) $k = 0$ ***b)*** $k = 1$ ***c)*** $k = 2$ ***d)*** $k = 3$

```
let rewriteN (t : term) ~(rules : rule list) : term * int =
  let rec doit (t : term) (step : int) =
    match rewrite1 t rules with
    | None   -> (t, step)
    | Some t -> doit t (step + 1)
  in
  doit t 0
```

Question à développer pendant l'oral 5 *Donner la complexité en temps et borner la taille de la sortie de* `rewriteN t rules` *en supposant l'ensemble des règles fixé. On ne fera aucune hypothèse de linéarité sur les règles utilisées.*

Complexité de `rewriteN t rules` si les règles sont L-linéaire à droite, en supposant l'ensemble de règles fixé.

Borne sur la taille de la sortie ($L = 1$) : $|\texttt{t}| + \texttt{n} \cdot K$
— chaque réécriture ajoute au plus un nombre constant de noeuds

Complexité en temps ($L = 1$) : $\mathcal{O}(\texttt{n} \cdot (|\texttt{t}| + \texttt{n})^3)$
— en utilisant le fait que le i-ème terme dans la séquence de réécriture est de taille borné par $|\texttt{t}| + i \cdot K$ (pour une certaine constante K), on borne la complexité par :

$$\sum_{i \leq n} M \cdot (|\texttt{t}| + i \cdot K)^3 \leq M \cdot \texttt{n} \cdot (|\texttt{t}| + \texttt{n} \cdot K)^3 \leq M \cdot K^3 \cdot \texttt{n} \cdot (|\texttt{t}| + \texttt{n})^3$$

Borne sur la taille de la sortie ($L > 1$) : $C^{\texttt{n}} \cdot |\texttt{t}|$
— chaque réécriture multiplie la taille du terme par au plus C
Complexité en temps ($L > 1$) : $\mathcal{O}(\texttt{n} \cdot C^{3 \cdot \texttt{n}} \cdot |\texttt{t}|^3) = \mathcal{O}(C'^{\texttt{n}} \cdot |\texttt{t}|^3)$, donc exponentielle.

2 Partie 2 (C) : Table de hachage

Vous devez utiliser C tout au long de cette partie.

Code fourni Le fichier `base.c` contient les éléments de code suivant :
- un ensemble d'inclusions usuelles en en-tête (`<stdlib.h>`, `<stdio.h>`, ...)
- une fonction de génération de nombres pseudo-aléatoires `rands` et une fonction de hachage `hash` qui sont décrites plus bas.

Implémentation d'un dictionnaire Un dictionnaire est une structure de données abstraite permettant de stocker un ensemble de liaisons de clé/valeur `(k,v)` telle que chaque clé `k` soit unique. Typiquement, il est attendu d'une structure de dictionnaire qu'elle supporte les opérations suivantes de façon efficace :
- `search(d,k)` : cherche la valeur `v` associée à la clé `k` dans le dictionnaire `d`.
- `add(d,k,v)` : ajoute la liaison clé/valeur `(k,v)` dans le dictionnaire `d`. Si une liaison `(k,v0)` était déjà présente pour la clé `k`, elle est remplacée par la nouvelle liaison `(k,v)`.
- `remove(d,k)` : supprime la liaison de clé `k` dans le dictionnaire `d`.

Nous allons implémenter un dictionnaire à l'aide d'une table de hachage par sondage linéaire. Nous considérerons des clés et valeurs de type `uint32_t`, et utiliserons la fonction de hachage fournie. Celle-ci est de signature `uint32_t hash(uint32_t key, uint32_t N)`, permet de hacher une clé `key` pour le paramètre `N` et est telle que `hash(key, N)` $\in \{0;\ldots;N-1\}$ pour tout `key, N`.

Le hachage par sondage linéaire fonctionne comme suit. On va stocker un ensemble de `n` liaisons clé/valeur `(k1,v1), ..., (kn,vn)` dans un tableau de taille `N` fixée. Dans un premier temps, on supposera le tableau suffisamment grand pour pouvoir stocker toutes les liaisons, c'est-à-dire qu'on demande que $n \leq N$. En temps normal, toutes les cellules du tableau ne sont pas remplies, et on aura donc $n < N$.
Par défaut, une liaison `(k,v)` est stockée dans la case d'indice `hash(k,N)` du tableau. Par exemple :

```
N = 6
Liaisons : (k1,v1), (k2,v2)
 hash(k1,N) = 5
 hash(k2,N) = 1
```

indice	hash	clé	valeur
0			
1	1	k2	v2
2			
3			
4			
5	5	k1	v1

Cependant, lors de l'ajout d'une liaison `(k,v)`, il est possible qu'une valeur soit déjà présente à la case d'indice `hash(k,N)` (on dit qu'il y a une collision). Dans ce cas, on stocke la liaison `(k,v)` dans la première case libre disponible à partir de l'indice `hash(k,N)`. Par exemple :

```
N = 6
Liaisons : (k1,v1), (k2,v2), (k3,v3), (k4,v4)
 hash(k1,N) = 5
 hash(k2,N) = 1
 hash(k3,N) = 1 /* collision (k2), déplacée indice 2 */
 hash(k4,N) = 2 /* collision (k3), déplacée indice 3 */
```

indice	hash	clé	valeur
0			
1	1	k2	v2
2	1	k3	v3
3	2	k4	v4
4			
5	5	k1	v1

Les indices du tableau doivent être considéré modulo sa taille `N`. Par exemple, l'ajout d'une liaison `(k,v)` de hache `5` dans le tableau ci-dessus insérera celle-ci dans la case suivant la case

d'indice 5 (car celle-ci est occupée), c'est à dire dans la case d'indice 0. Enfin, on notera que le tableau obtenu peut dépendre de l'ordre d'ajout des liaisons dans celui-ci.

On appelle surcharge d'une table de hachage par sondage linéaire `d` de taille `N`, et l'on note `surcharge(d)`, le nombre de liaisons `(k,v)` stockées dans la table à une autre position que `hash(k,N)`. Ainsi, la table ci-dessus a une surcharge de 2, à cause des liaisons `(k3,v3)` et `(k4,v4)`.

Question 6 *Implémenter une structure de donnée réalisant une table de hachage par sondage linéaire, l'opération d'ajout* `add(d,k,v)`*, et la fonction* `surcharge(d)`*.*
(On ne demande pas d'implémenter les fonctions `search` *et* `remove`*.)*
Créer une table de hachage `d` *de taille* `N`*, et utiliser la fonction* `rands` *qui vous est fournie pour calculer la surcharge de la table de hachage* `d` *calculée par le programme suivant :*

```
uint32_t *r = rands(u0,len);

for (int i = 0; i < len; i++){
  uint32_t key = r[i] % (10 * len);
  uint32_t val = (17 * (r[i] + i)) % 20;
  add(d,key,val);
}
```

pour `u0` $= u_0$ *et* `N` $= 4 \times$ `len`*, et cela pour les valeurs de* `len` *suivantes :*

a) `len` $= 100$ ***b)*** `len` $= 10\,000$ ***c)*** `len` $= 1\,000\,000$ ***d)*** `len` $= 10\,000\,000$

```
/** une entré dans une table de hachage */
struct record_t {
  uint32_t key;
  uint32_t value;
  uint32_t hash;
  bool empty;              /* true: empty; false: used */
};
typedef struct record_t record;

/* ------------------------------------------------------------------ */
/** une table de hachage */
struct htable_t {
  uint32_t max_size;        /* taille maximum de la table de hachage */
  uint32_t size;            /* taille actuelle de la table de hachage*/
  bool resizable;           /* est-ce que la table peut être rehaché */
  record *data;             /* pointeur vers un tableau de [max_size] case */
};
typedef struct htable_t htable;
```

```c
record* create_records(uint32_t max_size){
  record *data = malloc(max_size * sizeof(record));

  /* toutes les cellules sont initialement vide */
  for(uint32_t i=0; i < max_size; i++){
    data[i].empty = true;
  }

  return data;
}

/* ------------------------------------------------------------------ */
htable* create(uint32_t max_size, bool resizable){
  htable *h = malloc(sizeof(htable));
  record *data = create_records(max_size);

  h->max_size=max_size;
  h->size=0;
  h->data = data;
  h->resizable = resizable;
  return h;
}
```

```c
void add(htable *tbl, uint32_t key, uint32_t val){
  uint32_t h = hash(key, tbl->max_size);
  uint32_t i = 0;
  uint32_t cell;
  record *r;

  /* on cherche la première cellule vide ou cellule contenant `key` à
     partir de `h` */
  while(i < tbl->max_size){
    cell = (h + i) % tbl->max_size;
    r    = &(tbl->data[cell]);
    if (r->empty     ) { tbl->size++; break; }
    if (r->key == key) {              break; }
    i++;
  }

  if(i == tbl->max_size) { abort (); }  /* on crash si la tbl est pleine */

  r->key   = key;
  r->value = val;
  r->hash  = h;
  r->empty = false;
}
```

Question à développer pendant l'oral 6 *Décrire votre choix de structure de données. Expliquer informellement comment évolue la complexité en temps de l'ajout d'un élément dans la table en fonction du nombre de liaisons déjà présentes dans celle-ci.*

La structure `record` sert à stocker une liaison `(key,value)` de hache `hash`. Le champ booléen `empty` indique que l'entrée est libre et ne contient pour l'instant aucune valeur.

La table de hachage est représenté par la structure `htable`. Il s'agit d'un tableau de `max_size` cellules, dont `size` cellules contiennent une liaison.
Le booléen `resizable` sert pour la seconde question de cette partie.

Plus la table est pleine, plus l'ajout d'un élément est lent car il est peut être nécessaire d'essayer un grand nombre de cellules consécutives avant d'en trouver une disponible.

Question à développer pendant l'oral 7 *Décrire l'algorithme que vous utiliseriez pour implémenter les fonctions* `search` *et* `remove` *décrites au début de cette partie (on ne demande pas de réellement les implémenter).*

Pour `search(k,d)`, on scanne linéairement la table au départ de `hash(k,d.max_size)`, jusqu'à trouver une cellule contenant la clé `k`, ou une cellule vide (donc telle que `empty` soit à `true`).

Pour `delete(k,d)`, c'est plus complexe.
- Tout d'abord, on cherche l'indice `i` de la cellule contenant la clé `k` comme décrit ci-dessus.
- Ensuite, on libère cette cellule en mettant son booléen à `true`.
- Enfin, il faut re-compacter la table à partir de l'indice `i`. Pour cela, on parcours les cellules à partir de l'indice `j = i+1`. Si la cellule `j` est vide, on s'arrête. Sinon, on vérifie si la liaison dans la cellule `j` devrait être placée en cellule `i`, c'est à dire si son hache est plus petit que `i` (en gérant correctement l'aspect cyclique du tableau, ce qui n'est pas évident) :
  - si c'est le cas, on déplace la cellule `j` en `i`, et on reprend la procédure pour re-compacter le tableau à partir de `j`.
  - sinon, on passe à la cellule suivante en incrémentant `j`.

Le taux de remplissage τ d'une table de hachage est le ratio du nombre de cellules utilisées sur le nombre de cellules total de la table. Lorsque celui-ci devient trop grand, il est nécessaire d'augmenter la taille de la table de hachage en passant d'une table de taille `N` à une table de taille `2 * N` : on dit qu'on re-dimensionne la table. Concrètement, si `d` est une table de taille `N`, on crée une nouvelle table de taille `2 * N`, dans laquelle on transfère tous les éléments qui étaient présents dans `d`. On fera ce transfert élément par élément, en parcourant l'ancienne table dans l'ordre des indices croissants. On note `resize(d)` cette opération. Bien sûr, lors de l'ajout à la nouvelle table de taille `2*N`, les éléments sont de nouveau hachés, à l'aide la fonction `hash(., 2*N)` et non plus avec `hash(., N)` comme précédemment.
Une stratégie consiste à re-dimensionner la table à chaque fois que son taux de remplissage dépasse strictement 25%. Par exemple, si $N = 16$ et que 4 liaisons sont déjà dans la table, celle-ci sera re-dimensionnée lors de l'ajout d'une 5^{e} liaison. On note `add_resize(d,k,v)` la fonction qui se comporte comme `add(d,k,v)`, à ceci près qu'elle re-dimensionne la table `d` si nécessaire après l'ajout de la liaison `(k,v)`.

Question 7 *Implémenter l'opération* `add_resize(d,k,v)`.
Créer une table de hachage `d` *de taille initiale* `16`*, et utiliser la fonction* `rands` *fournie pour calculer la surcharge de la table de hachage* `d` *calculée par le programme suivant :*

```
uint32_t *r = rands(u0,len);

for (int i = 0; i < len; i++){
  uint32_t key = r[i] % (10 * len);
  uint32_t val = (17 * (r[i] + i)) % 20;
  add_resize(d,key,val);
}
```

pour `u0` $= u_0$ *et* $N = 4 \times$ `len`*, et cela pour les valeurs* `len` *suivantes :*

a) `len` $= 100$ ***b)*** `len` $= 10\,000$ ***c)*** `len` $= 1\,000\,000$ ***d)*** `len` $= 10\,000\,000$

```
void rehash(htable *tbl, uint32_t new_max_size){
  record *new_data = create_records(new_max_size);
  record *old_data = tbl->data;
  uint32_t old_max_size = tbl->max_size;

  tbl->max_size=new_max_size;
  tbl->data = new_data;
  tbl->size=0;

  /* toutes les cellules sont initialement vide */
  for(int i=0; i < old_max_size; i++){
    if (!old_data[i].empty) {
      add(tbl, old_data[i].key, old_data[i].value);
    }
  }

  free(old_data);
}

void add_resize(htable *tbl, uint32_t key, uint32_t val){
  add(tbl,key,val);

  if (tbl->resizable && tbl->size > tbl->max_size/4) {
    rehash(tbl, tbl->max_size * 2);
  }
}
```

3 Partie 3 (OCaml) : Réécriture efficace

Vous devez utiliser OCaml tout au long de cette partie.

Code fourni Cette partie suit la Partie 1, et utilisera le même fichier de base `base.ml`.

Terme abstrait et représentation Dans cette partie, on fera la différence entre un terme abstrait et sa représentation par un terme OCaml. Par exemple, le terme abstrait `z` peut être représenté par le terme OCaml `Tfun(Tag.default,"z",[])`. Un même terme abstrait peut avoir

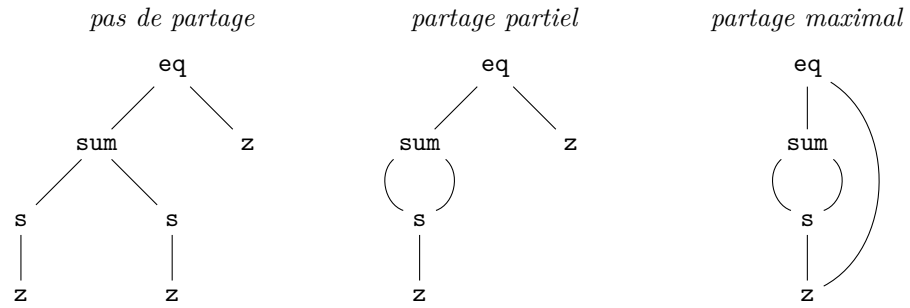
plusieurs représentations par des termes OCaml, de façon plus ou moins efficace en mémoire. Par exemple, voici deux représentations du terme `eq(z,z)` :

```
let d = Tag.default in                    let d = Tag.default in
Tfun(d,"eq",[Tfun(d,"z",[]);              let z = Tfun(d,"z",[]) in
             Tfun(d,"z",[])])             Tfun(d,"eq",[z; z])
```

La représentation de droite utilise un espace mémoire réduit car le sous-terme `z` est partagé : en effet, celui-ci n'est construit qu'une seule fois à droite par la liaison `let z = ...`.

3.1 Partage maximal

On dit qu'un terme OCaml `t` de type `term` est *hash-consé* s'il a la propriété de partage maximal, c.-à-d. si dès que deux sous-termes de `t` sont égaux, ils sont représentés **physiquement dans la mémoire** par le même objet OCaml. Par exemple, voilà trois représentations possibles (sous forme de graphes) du même terme abstrait `eq(sum(s(z),s(z)),z)` :



Nous allons utiliser la technique suivante pour construire des termes OCaml *hash-consés*.

Tags Chaque terme *hash-consé* `t = Tfun(tag,s,args)` aura un tag `tag` (différent de `Tag.default`). Le tag d'un terme caractérisera de façon unique celui-ci :

- les termes *hash-consés* représentant des termes abstraits différents auront des tags différents ;
- les termes OCaml représentant le même terme abstrait seront physiquement égaux, et auront donc (entre autres) le même tag.

Hash-consing On maintient l'ensemble des termes *hash-consés* déjà construits dans une structure de données $\mathcal{H}$ (détaillée par la suite). Pour construire le terme *hash-consé* `f(args)`, où les sous-termes `args` sont déjà *hash-consés*, on cherche si `f(args)` est déjà présent dans $\mathcal{H}$: si c'est le cas, on réutilise le terme OCaml dans $\mathcal{H}$; sinon, on créé un nouveau tag `tag` frais à l'aide de `Tag.create()`, on stocke `Tfun(tag,f,args)` dans $\mathcal{H}$ et on renvoie ce nouveau terme.

Structure $\mathcal{H}$ La structure de donnée $\mathcal{H}$ et la fonction `fun_hashcons` permettant de la manipuler sont décrites ci-dessous. Celles-ci sont **déjà fournies** dans le module `HashCons` du fichier `base.ml`. **Il ne faut pas les ré-implémenter ou les recopier !**

On implémente la structure $\mathcal{H}$ par une table de hachage ayant le type enregistrement suivant :

```
type hashtbl : { table : term list array; size : int }
```

où `size` est la taille du tableau `table`, et la cellule en position `h` du tableau `table` contient la liste `l` des termes *hash-consés* de hache `h`, où ce hache est obtenu à l'aide de la fonction fournie `hash_for_hashconsing`.

Enfin, le module `HashCons` définit une seule fonction de signature :

```
val fun_hashcons : string -> term list -> term
```

telle que `(fun_hashcons s args)` calcule le terme *hash-consé* `s(args)`, en supposant que les termes `args` sont **déjà hash-consés**.

Question à développer pendant l'oral 8 *Regarder le code du module* `HashCons` *et répondre aux questions suivantes :*

- *Quel est l'intérêt d'utiliser la fonction de hachage* `hash_for_hashconsing` *plutôt que la fonction de hachage* `hash` *de la Question 1 ?*
- *Pourquoi utiliser la fonction* `equal_for_hashconsing` *plutôt que la fonction d'égalité usuelle = ? Pourquoi cette fonction est-elle correcte ?*
- *Comment la fonction* `fun_hashcons` *réalise-t-elle sa spécification ?*

Supposons que le nombre maximum d'arguments d'un symbole de fonction est fixé, ainsi que la taille des symboles de fonctions.

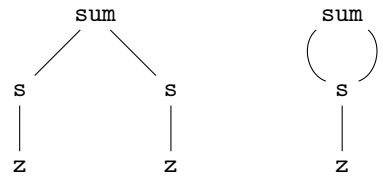
- Dans ce cas, la fonction `hash_for_hashconsing` est en temps constant, contrairement à la fonction `hash` qui est en temps linéaire.
  Cela exploite le fait que les sous-termes strictes sont déjà hash-consés, et donc uniquement caractérisé par leur tag, ce qui donne un excellent hache calculable en temps constant.
- Idem pour le test d'égalité :
  - L'égalité structurelle d'Ocaml `t=t'` est en temps linéaire en la taille de `t` et `t'` (plus précisément en $\mathcal{O}(\min(|\texttt{t}|, |\texttt{t'}|))$.
  - Le test d'égalité `equal_for_hashconsing t t'` est seulement correcte si les sous-termes strictes de `t` et `t'` sont déjà hash-consé, auquel cas le test d'égalité pour les sous-termes stricte peut être fait à l'aide de leur tag.
    Cela donne un teste d'égalité en temps constant, bien meilleur que le teste d'égalité structurelle.
- La fonction `fun_hashcons f args` utilise le fait que les arguments `args` sont déjà hash-consé pour garantir que `hash_for_hashconsing` et `equal_for_hashconsing` sont correctes et en temps constant.
  Tant que l'ajout dans la table de hachage est en rapide (ce qui sera le cas si il y a peu de collisions), cela donne une fonction très efficace.

Soit `t` un terme OCaml. La taille de `t`, notée $\mathsf{taille}(t)$, est la taille du terme abstrait représenté par `t`, c.-à-d. :

$$\mathsf{taille}(f(t_1 \ldots, t_n) = 1 + \sum_{i=1}^{n} \mathsf{taille}(t_i)$$

La taille en mémoire de `t` est la taille de la représentation en mémoire de `t`, c.-à-d. le nombre de

nœuds du graphe représentant `t`. Par exemple, les deux représentations suivantes :



du même terme abstrait $t = \texttt{sum}(\texttt{s}(\texttt{z}), \texttt{s}(\texttt{z}))$ sont toutes deux de taille 5, mais celle de gauche est de taille en mémoire 5, alors celle de droite est de taille en mémoire 3.

Question 8 *Implémenter la fonction* `hashcons` *de type* `term -> term` *telle que* `hashcons t` *calcule le hash-consé de* `t`.

Implémenter les fonctions `size` *et* `size_mem` *de type* `term -> int` *telles que* `size t` *calcule la taille du terme* `t`, *et* `size_mem t` *calcule la taille en mémoire d'un terme* `t` *(que l'on supposera* ***hash-consé****).*

Notons `terms = read_terms "data/U0/q8.txt"`.
Soit `t` *le k-ième terme de* `terms`. *Calculer le couple* `(size t', size_mem t')`, *où* `t' = hashcons t` *pour les valeurs de k suivantes :*

a) $k = 0$ ***b)*** $k = 1$ ***c)*** $k = 2$ ***d)*** $k = 3$

```
(** compte le nombre de noeud dans un terme hash-consé *)
let size_mem (t : term) : int =
  (* ensemble des terms déjà vu, stocké par leur tag *)
  let seen = ref [] in
  let cpt = ref 0 in

  let rec doit t : unit =
    match t with
    | Tfun (tag, _, terms) ->
      assert (tag <> Tag.default);
      let tag = Tag.to_int tag in
      (* on ne recompte pas un noeud déjà compté *)
      if not (List.mem tag !seen) then
        begin
          seen := tag :: !seen;
          incr cpt;
          List.iter doit terms
        end;
  in
  doit t;
  !cpt
```

Question à développer pendant l'oral 9 *Détailler votre implémentation de* `size_mem` *et donner sa complexité en temps.*

Il suffit de maintenir dans une structure de donnée annexe l'ensemble des sous-termes dont qu'on a déjà compté.
Il s'agit donc tout simplement d'implémenter une structure d'ensemble. Ici, il est plus efficace (et très simple) de maintenir l'ensemble des tags plutôt que l'ensemble des termes, car les tags permettent des testes d'égalité en temps constant. Cette structure peut être fait de différente façon :

— de façon naïve avec la liste des tags déjà compté (c'est suffisant dans ce sujet) ;
— de façon plus optimisé, par exemple avec une table de hachage ou des arbres binaires (équilibrés ou pas).

3.2 Réécriture et partage maximal

Question 9 *Réécrire la fonction* `rewriteN` *en supposant que le terme de départ est hash-consé, et en garantissant que le terme calculé l'est aussi. On appellera cette nouvelle implémentation* `rewriteN_hashconsed`.

Indications. *Copier-coller votre code précédent, modifier attentivement celui-ci pour garantir que des termes hash-consés sont produits, et optimiser le code en exploitant le hash-consing.*

Attention, une implémentation trop naïve de la fonction `size_mem` *ne sera pas capable de gérer les plus grands termes construits dans cette question.*

Notons `terms = read_terms "data/U0/tq9.txt"` *et* `rules = read_rules "data/U0/rq9.txt"`. *Soit* `t` *le k-ième terme de* `terms`. *Calculer le couple* `(size_mem t', len)` *telle que* `rewriteN_hashconsed t rules = (t', len)` *pour les valeurs de k suivantes :*

a) $k = 0$ ***b)*** $k = 1$ ***c)*** $k = 2$ ***d)*** $k = 3$

```
let get_tag (t : term) = match t with Tfun (tag,_,_) -> tag

(** comme [find_match], sauf qu'on utilise le test d'égalité
    rapide à la place de l'égalité structurelle de Ocaml. *)
let find_match_hashconsed (t1 : term) (t2 : rule_term) : subst option =
  let exception NoMatch in
  let no_match () = raise NoMatch in

  let s = Array.make max_var None in

  let rec doit t1 t2 =
    let Tfun (_,f1,l1) = t1 in
    match t2 with
    | Rfun (f2, l2) ->
      if f1 <> f2 then no_match ();
      List.iter2 doit l1 l2

    | Rvar v ->
      match s.(v) with
      | None   -> s.(v) <- Some t1
      | Some t ->
        (* on fait attention à bien utiliser le test d'égalité rapide *)
        if get_tag t <> get_tag t1 then no_match ()
  in
  try doit t1 t2; Some s with NoMatch -> None
```

```
(** comme [rewrite1], avec deux changements:
   - on utilise [find_match_hashconsed] à la place de [find_match]
   - on garde dans une structure de set l'ensemble des tags
     des termes déjà traités, pour ne pas les reconsidérer. *)
let rewrite1_hashconsed (t : term) (rules : rule list) : term option =
  let seen = Hashtbl.create 100000 in
  let found = ref false in

  (* cherche à appliquer une règle dans [t] *)
  let rec doit t =
    if !found || Hashtbl.mem seen (get_tag t) then t else begin
      Hashtbl.add seen (get_tag t) ();
      (* on cherche une règle s'appliquant en tête *)
      match t with
      | Tfun (_, f, l) as t ->
        let t = (* itére sur les règles *)
          List.find_map (fun (rl, rr) ->
              match find_match_hashconsed t rl with
              | None   -> None

              (* [s] tel que [t = subst s rl], on renvoie [subst s rr] *)
              | Some (s : subst) -> Some (subst s rr)
            ) rules
        in
        match t with
        | Some t -> found := true; t   (* match trouvé en position de tête *)

        (* pas de match en position de tête dans [t], on récurse *)
        | None -> HashCons.fun_hashcons f (List.map doit l)
    end
  in
  let t = doit t in
  if !found then Some t else None
```

```
let rewriteN_hashconsed (t: term) (rules: rule list) : term*int =
  let rec doit (t : term) (step : int) =
    match rewrite1_hashconsed t rules with
    | None   -> (t, step)
    | Some t -> doit t (step + 1)
  in
  doit t 0
```

Question à développer pendant l'oral 10 *Donner la complexité en temps et borner la taille de la sortie de* `rewriteN_hashconsed t rules n` *en supposant l'ensemble des règles fixé, et que les termes de règles à droites des règles sont tous au plus L-linéaire.*

Le matching de `t` et `l` se fait en temps $\mathcal{O}(\min(|\texttt{t}|, |\texttt{l}|) = \mathcal{O}(1)$ puisque l'ensemble des règles est supposé fixé.

La réécriture d'une des règles de `rules` dans un terme `t` de taille en mémoire T se fait en temps $\mathcal{O}(T)$, et donne un terme de taille $T + C$ ou C est une constante qui dépend seulement de l'ensemble des règles `rules`. Noter que cela peut être exponentiellement plus rapide que la réécriture sans hash-consing, puisqu'on peut avoir $|\texttt{t}| \approx 2^T$.

La réécriture de `n` règles de `rules` dans un terme `t` de taille en mémoire T se fait en temps $\mathcal{O}(\texttt{n} \cdot T)$, et donne un terme de taille $T + \texttt{n} \cdot C$.

On rappelle que la longueur d'une séquence est le nombre de réécritures dans celle-ci.

Question 10 *Implémenter la fonction de signature :*

```
val longest : term -> rule list -> int
```

telle que `longest t rules` *calcule la longueur de la* ***plus longue*** *séquence de réécritures au départ de* `t` *par* `rules`*. On considérera toutes les séquences de réécritures possibles, et pas seulement celles obtenues par* `rewrite1` *: en particulier, l'ordre des règles dans* `rules` *n'a plus d'importance, et on ne s'arrêtera pas à la première réécriture rencontrée lors d'un parcours en profondeur. On supposera que toutes les séquences de réécritures au départ de* `t` *sont de longueur finie.*

Notons `terms = read_terms "data/U0/tq10.txt"` *et soit* `t` *le k-ième terme de* `terms`*.*
Notons `rules = read_rules "data/U0/rq10.txt"`*.*
Calculer `longest t rules` *pour les valeurs de k suivantes :*

a) $k = 0$ ***b)*** $k = 1$ ***c)*** $k = 2$ ***d)*** $k = 3$

```
let rewrite1_all (t : term) (rules : rule list) : term list =
  let rec doit t : term list =
    let Tfun (_, f, args) = t in
    let terms_head =
      (* itére sur les règles *)
      List.filter_map (fun (rl, rr) ->
          match find_match_hashconsed t rl with
          | None   -> None

          (* [s] tel que [t = subst s rl], on stocke [subst s rr] *)
          | Some s -> Some (subst s rr)
        ) rules
    in
    (* on récurse pour trouver des matches dans les sous-termes de [t] *)
    let terms_sub =
      let list_args' = doit_sub ~left:[] ~right:args ~acc:[] in
      List.map (fun args' -> HashCons.fun_hashcons f args') list_args'
    in
    terms_head @ terms_sub

  (* [left] à l'envers *)
  and doit_sub
      ~(left : term list) ~(right : term list) ~(acc : term list list)
    : term list list
    =
    match right with
    | [] -> acc
    | arg :: right ->
      let list_arg' = doit arg in
      let acc =
        List.fold_left (fun acc arg' ->
            let args' = List.rev_append left (arg' :: right) in
            args' :: acc
          ) acc list_arg'
      in
      doit_sub ~left:(arg :: left) ~right ~acc
  in
  (* Table de hachage utilisée comme un set pour enlever les doublons.
     On aurait aussi pu enlever les doublons lors des appels récursifs. *)
  let hashtbl = Hashtbl.create 65536 in
  List.iter (fun x -> Hashtbl.add hashtbl (get_tag x) x) (doit t);
  Hashtbl.fold (fun _ t acc -> t :: acc) hashtbl []
```

```ocaml
let longest (t : term) (rules : rule list) : int * term list =
  (* la longueur du plus grand chemin au départ d'un terme *)
  let hashtbl : (_, int * term list) Hashtbl.t = Hashtbl.create 65536 in

  let rec longest (t : term) : int * term list =
    try Hashtbl.find hashtbl (get_tag t) with
    | Not_found ->
      let nexts = rewrite1_all t rules in
      (* on a [t → t'] *)
      let max, path =
        (* séquence la plus longue au départ d'un successeur *)
        List.fold_left (fun (max,path) t' ->
            let len', path' = longest t' in
            if max < len' then (len', path') else (max, path)
          ) (0, []) nexts
      in
      let max, path = max + 1, t :: path in
      Hashtbl.add hashtbl (get_tag t) (max, path);
      (max, path)
  in
  let max, path = longest t in

  (* la nombre de réécriture est la longueur du chemin - 1 *)
  max - 1, path
```

Question à développer pendant l'oral 11 *Décrire votre implémentation de la fonction* `longest`. *Exprimer sa complexité en temps en fonction, entre autres, de la longueur L de la plus longue séquence de réécritures au départ de* `t`. *On supposera l'ensemble des règles fixé.*

On implémente une fonction `longest t'` calculant la longueur de la plus longue séquence au départ de `t`. Cette fonction est **mémoïsé**, en enregistrant dans un structure de donnée (liste d'association, table de hachage, tableau redimensionnable) les liaisons `(tag, l)` ou `tag` est le tag d'un terme hash-consé `t'` tel que `longest t' = l`. Ainsi, chaque terme `t'` accessible depuis `t` est traité par `longest` une seule fois.

On va maintenant faire une analyse grossière de la complexité de notre algorithme.

Un terme `t'` a au plus $C \cdot |\texttt{t'}|$ successeurs par $\rightarrow_{\texttt{rules}}$, ou C est une constante (puisque l'ensemble des règles est supposé fixé). Pour borner la complexité de `longest t`, on va borner le nombre de termes accessibles au départ de `t` est leur taille :

- Un terme `t'` obtenue en au plus L réécritures à partir de `t` est de taille au plus $C^L \cdot |\texttt{t}|$ ou C est une constante.
- On a donc un arbre dont chaque noeud a au plus $C^L \cdot |\texttt{t}|$ successeurs, et de profondeur L. On a donc au plus $C^{L^2} \cdot |\texttt{t}|^L$ termes accessibles.

La complexité d'un appel `longest t'`, en excluant les appels récursifs, est linéaire en $|\texttt{t'}|$, donc en $\mathcal{O}(C^L \cdot |\texttt{t}|)$.

On a donc une complexité totale en $\mathcal{O}(C^{L^2} \cdot |\texttt{t}|^L \cdot C^L \cdot |\texttt{t}|) = \mathcal{O}(C^{L^2+L} \cdot |\texttt{t}|^{L+1})$.

Fiche réponse type : Réécriture et partage

$\widetilde{u_0}$: 1

Question 1

a) 73085

b) 79891

c) 32362

d) 562

Question 2

a) 64609

b) 21394

c) 81884

d) 15741

Question 3

a) Some 128585

b) None

c) Some 49680

d) None

Question 4

a) Some 79136

b) Some 16665

c) Some 820

d) None

Question 5

a) 2

b) 6

c) 52

d) 1679

Question 6

a) 9

b) 1092

c) 112930

d) 1130432

Question 7

a) 4

b) 755

c) 108190

d) 667827

Question 8

a) (8,6)

b) (45,24)

c) (195,82)

d) (695,282)

Question 9

a) (8,22)

b) (10,37)

c) (135,1228)

d) (263,2572)

Question 10

a) 5

b) 7

c) 18

d) 22