BANQUE MP INTER-ENS – SESSION 2023
RAPPORT SUR L'ÉPREUVE PRATIQUE D'ALGORITHMIQUE ET DE PROGRAMMATION DU CONCOURS COMMUN DES ÉCOLES NORMALES SUPÉRIEURES

Écoles concernées : Lyon, Paris-Saclay, Rennes, Ulm

Coefficients (en pourcentage du total d'admission)
- Lyon : 16,9 %
- Paris-Saclay : 13,2 %
- Ulm : 13,3 %

Jury : Adrien Koutsos, Joseph Lallemand, Gabriel Radanne, Yann Ramusat

Contenu de ce document

Dans ce rapport, après avoir rappelé l'organisation de l'épreuve, nous faisons quelques remarques générales sur son déroulement. 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, leurs implémentations, 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 23 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 typiquement générées pseudo-aléatoirement, à partir d'une suite pseudo-aléatoire initialisée par une valeur u_0, différente pour chaque personne, distribuée au début de l'épreuve. Éventuellement certaines données peuvent être lues dans des fichiers fournis.

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, typiquement numériques, bien 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 typiquement 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 du u_0 de la leur. 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 50 % de la note finale. On observe dans l'ensemble, quoique pas systématiquement, une bonne corrélation entre les résultats obtenus aux deux parties.

Lecture de fichiers ou de l'entrée standard. Depuis 2022, dans certains sujets, une nouveauté a été introduite dans l'épreuve : des données d'entrée sont parfois lues depuis des fichiers, plutôt que générées aléatoirement. Dans ce cas, les fichiers contenant ces données sont fournis aux candidats et candidates au début de l'épreuve, ainsi que du code permettant de les lire. Il peut par exemple être demandé d'utiliser ce code pour récupérer une partie d'un fichier fourni dépendant du u_0, qui sera utilisée comme donnée d'entrée – nous renvoyons le lecteur aux sujets de MPI de cette année qui en sont des exemples.

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 qui avaient soigné la lisibilité de leur code, et seulement si les réponses aux questions étaient 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 sont communes entre les années. 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 avec une version du code pour u_0 et

une pour $\widetilde{u_0}$, et plutôt de lire le u_0 dans une variable et d'exécuter tout le code avec, *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 au tableau. Si le jury n'est pas convaincu par un argument simple, il sera toujours possible de le convaincre par 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 aborder des questions globalement peu 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 ici 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.

Sur 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. L'écueil inverse, de ne pas utiliser du tout le tableau est plus rare, mais il rend parfois le raisonnement difficile 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 ou un exemple au tableau quand une explication s'y prête, ensuite de faire la preuve dessus à l'oral. Dans le cas d'un raisonnement 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 semblé au jury que certaines personnes n'ont pas osé résoudre certaines 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.

Langages de programmation. En MPI, les sujets demandent explicitement d'utiliser les langages OCaml et C, sans laisser le choix. Les candidats et candidates nous ont semblé moins à l'aise en C qu'en OCaml : il est important de s'entraîner avec les deux langages, qui sont tous deux exigés.

Les sujets de MP, quant à eux, sont prévus et calibrés pour être de difficulté équivalente dans les langages Python et OCaml. Nous notons 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. Certaines années, 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 voulons aussi conseiller d'étudier les structures de données basiques pour le ou les langages utilisés. La différence en efficacité d'un programme qui utilise une liste la où il aurait fallu un tableau, ou l'inverse, est très visible dans ce type de sujet. 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. 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 : Charge de cavalerie.

Ce sujet portait sur l'utilisation d'algorithmique des graphes pour résoudre des problèmes liés au déplacement de cavaliers sur un échiquier. Il se décomposait en deux parties. La première portait sur le problème d'un cavalier, restreint à ne se déplacer qu'en avant, qui souhaite maximiser la valeur des pièces capturées sur son chemin d'un bord de l'échiquier à l'autre. Ce problème était vu comme un problème de plus court chemin. Dans la seconde partie, on considérait le cas de multiples cavaliers, qui doivent coordonner leur attaque pour maximiser la valeur totale des pièces capturées en un coup, que l'on voyait comme une instance du problème de flot maximal.

La partie écrite commençait, classiquement, par les questions Q1 à Q4, portant sur la génération des données de test. Ces questions relativement simples ont été globalement bien réussies. Un point notable est le choix de la structure de données utilisée, qui intervenait dès la Q4. Idéalement, il fallait opter pour une structure permettant efficacement à la fois d'obtenir la valeur d'une case donnée, et d'itérer sur la liste des pièces – par exemple, un tableau stockant les valeurs, associé à une liste des positions, bien que d'autres choix fussent possibles. Le sujet, en particulier la question d'oral QO2, aiguillait les candidats et candidates dans cette direction. La Q5 demandait de calculer les mouvements possibles d'un cavalier, et était sans surprise. On proposait ensuite de calculer le score maximal pour un cavalier de trois manières. Tout d'abord, la Q6 demandait de procéder à une exploration exhaustive des chemins. Comme c'est généralement le cas dans les questions de recherche exhaustive, on n'attendait pas ici d'optimisation particulière : les valeurs numériques demandées étaient assez faibles pour être traitées par une approche brutale. Cette question a été assez bien réussie. Ensuite, on donnait l'algorithme de Bellman-Ford pour le calcul du score maximal. Il fallait l'implémenter (Q6), puis l'étendre pour calculer un chemin atteignant ce score (Q7). Pour ce faire, on pouvait maintenir une table indiquant le prédécesseur de chaque sommet : il fallait alors être attentif à mettre à jour le critère d'arrêt, pour s'assurer que cette table était elle aussi stabilisée – plusieurs personnes s'y sont trompées. Enfin, la Q9 demandait de formuler un algorithme plus efficace – on attendait un algorithme dynamique. La dernière partie a été abordée par peu de candidats et candidates, vraisemblablement par manque de temps. La Q10 demandait d'écrire un algorithme glouton, consistant à choisir tant que c'est possible le coup rapportant immédiatement le meilleur score. L'algorithme glouton naïf, qui considère à chaque fois tous les coups possibles, n'était pas assez efficace pour les valeurs numériques demandées : il fallait remarquer qu'on pouvait se contenter de calculer une seule fois la liste des coups triés par score au départ. Enfin, les Q11 à Q13 amenaient à implémenter une variante de l'algorithme de Ford-Fulkerson pour le calcul de flot maximal. Personne n'est parvenu à la fin de l'implémentation.

Pour la partie orale, mentionnons quelques points notables. La QO2 demandait de discuter le choix de la structure de données – on attendait que les candidats ou candidates sachent justifier leur choix, ou le critiquer s'ils ou elles s'étaient rendus compte trop tard qu'il était mauvais. Majorer la complexité du parcours exhaustif, en QO3, demandait de remarquer que la longueur des chemins étaient bornée par la hauteur de l'échiquier, et que le nombre total de chemins est exponentiel : trop de personnes se sont trompées sur ces points. La QO4 demandait de donner l'encodage du problème sous forme de graphe. Cette question était simple, mais on attendait une réponse détaillée : la définition précise du graphe, la bijection préservant le score entre chemins et marches. Les candidats et candidates ont bien réussi cette question dans l'ensemble. À la question QO5, on pouvait remarquer que la stabilisation se faisait en au plus n étapes, et obtenir ainsi une majoration en $O(n^3)$ plutôt que $O(n^4)$, quelques personnes l'ont vu. Pour un algorithme dynamique, à la QO7, on attendait la relation de récurrence utilisée. Cette question a été plutôt bien traitée, cependant peu ont su décrire l'optimisation en espace habituelle des algorithmes dynamiques consistant à ne garder que la ligne courante. Un certain nombre de candidats et candidates ont abordé la QO8 à l'oral, sans avoir eu le temps d'implémenter l'algorithme, et ont tout de même pu obtenir des points en évaluant sa complexité. La QO9 demandait un contre-exemple montrant que l'algorithme glouton n'était pas optimal – un tel cas figurait en fait dans l'exemple du sujet, certains et certaines l'ont repéré. Enfin, une seul personne a abordé la QO10.

Écrit	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13
Tous les points	100	82	94	79	88	61	39	24	9	6	3	0	0
Réponses partielles	100	94	100	88	94	64	52	33	12	9	3	0	0

Oral	QO1	QO2	QO3	QO4	QO5	QO6	QO7	QO8	QO9	QO10
Tous les points	67	64	55	73	85	15	36	9	18	0
Réponses partielles	100	100	100	91	91	67	45	36	24	3

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

SUJET 2 : DES MARCHES DANS DES GRAPHES.

Ce sujet abordait différents problèmes de calcul de marche de poids optimal dans des graphes. Le sujet commence de façon classique par la génération de graphe et leurs sommes de contrôle. La somme de contrôle est là pour vérifier que la procédure de génération des graphes des candidat · es est correcte. Ensuite, le sujet étudie plusieurs problèmes de marche dans des graphes, suivant diverses classes de stratégies. La première partie s'intéressait à la marche de concert de nombreux agents suivant une stratégie sans mémoire commune. Dans la seconde partie, un seul agent était considéré. Enfin, la dernière question étudiait des marches infinies.

Pour la partie écrite, les questions Q1 à Q5 étaient de simples exercices de programmation et ont été correctement traitées par presque tout · es les candidat · es. Les questions Q6 et Q7 demandaient de faire se déplacer simultanément un grand nombre d'agents sur des graphes. Bien que la majorité des candidat · es ait proposé une solution, réussir à résoudre les grands cas demandaient plusieurs optimisations pour factoriser les déplacements des agents, ce que peu de candidat · es ont réussi. Dans la question Q8, il s'agissait de faire se déplacer un seul agent suivant une stratégie sans mémoire pour un très grand nombre de pas. Résoudre totalement cette question totalement demandait d'exploiter le fait qu'un tel

agent finit nécessairement dans une boucle, permettant d'accélérer le calcul de sa marche. Peu de candidat · es l'ont remarqué. Dans les questions suivants, les candidat · es devaient trouver des stratégies optimales au départ d'un sommet donné, mais cette fois avec mémoire. La première approche, proposée dans la question Q9, demandait une exploration exhaustive. Bien que standard, un faible nombre de candidat · es ont abordé cette question, probablement par manque de temps. La question Q10 n'était presque pas guidée, et pouvait se résoudre efficacement par programmation dynamique, ce que seuls les meilleurs candidat · es ont vu. Enfin, la question Q11 se réduisait à chercher des cycles de poids moyen optimal. Une approche exhaustive était possible mais insuffisante pour résoudre les plus grand cas, qui demandait une approche plus efficace, aussi par programmation dynamique. Un seul candidat · e a réussi à totalement résoudre cette question.

Pour la partie orale, nous nous contentons de discuter quelques questions notables. La question QO2 a été bien traité dans l'ensemble, bien que le jury a été étonné que certains candidat · es ne sachent pas motiver leur choix de représentation des graphes par des listes ou matrices d'adjacences. Lors de l'analyse de complexité de la question QO4, quelques candidat · es ont considéré — probablement par inattention — pouvoir calculer l'union de deux listes en $\mathcal{O}(1)$. Pour obtenir une bonne complexité dans la question QO5, il était nécessaire d'utiliser une structure de données supportant des unions en $\mathcal{O}(1)$, par exemple en utilisant un arbre binaire (même non équilibré). Surprenamment, très peu de candidat · es l'ont vu. Dans leur réponse à la question QO8, tout les candidat · es n'ont pas pensé à majorer le nombre de chemins de longueur L au départ d'un sommet par d^L ou d est le degré maximal du graphe.

Écrit	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
Tous les points	97	100	97	94	97	38	25	31	25	13	3
Réponses partielles	100	100	100	100	100	84	81	84	31	25	13

Oral	QO1	QO2	QO3	QO4	QO5	QO6	QO7	QO8	QO9	QO10
Tous les points	94	78	75	69	13	34	63	13	9	3
Réponses partielles	100	100	100	94	94	84	69	59	22	9

TABLE 2. Pourcentages de réponses correctes et partielles à chaque question du sujet 2.

Charge de cavalerie

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

Durée de l'épreuve : 3 heures 30 minutes

Juin/Juillet 2023

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.

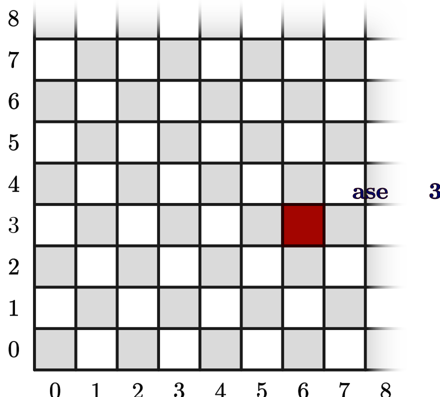
En ce qui concerne la partie orale de l'examen, lorsque la description d'un algorithme est demandée, vous devez présenter son fonctionnement de façon schématique, courte et précise. Vous ne devez en aucun cas recopier le code de vos procédures !
Quand on demande la complexité en temps ou en mémoire d'un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

1 Préliminaires

1.1 Échecs et cavaliers

Les échecs sont un jeu de stratégie opposant deux joueurs qui contrôlent respectivement des pièces noires et blanches. Aucune connaissance préalable sur le jeu d'échecs n'est requise. Nous précisons ici quelques-unes des règles du jeu, qui seront les seules importantes dans ce sujet.

Le jeu d'échecs se joue sur un plateau appelé l'échiquier, qui est une grille carrée de 8 cases de côté. Ces cases sont alternativement noires et blanches, mais leur couleur n'aura pas d'importance dans ce sujet. On considèrera ici plus généralement des échiquiers de taille $n \times n$, c'est-à-dire des grilles carrées de n cases de côté, où $n \in \mathbb{N}^\star$. On munit l'échiquier d'un système de coordonnées : à chaque case est associé un couple d'entiers $(i, j) \in [\![0, n-1]\!]^2$, appelé sa position, désignant respectivement sa colonne (abscisse) et sa rangée (ordonnée) [1], comme illustré dans l'exemple suivant où l'on a marqué la case $(6, 3)$.

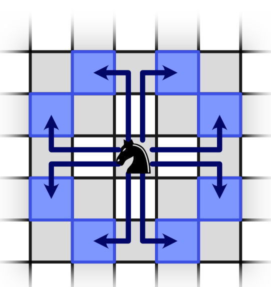


Sur un échiquier peuvent être placées et déplacées des pièces, qui peuvent être noires ou blanches. Le cavalier est l'une de ces pièces, symbolisée par ♘ (pour un cavalier blanc) ou ♞ (pour un cavalier noir). Son mouvement suit une forme de "L" : lorsque l'on joue un coup avec un cavalier, on peut le déplacer

— de 2 cases horizontalement et 1 case verticalement,

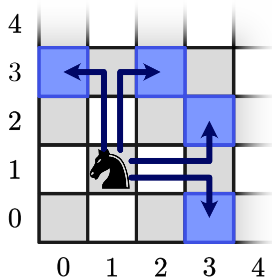
— ou bien de 2 cases verticalement et 1 case horizontalement.

Il en résulte en général 8 mouvements possibles pour un cavalier :

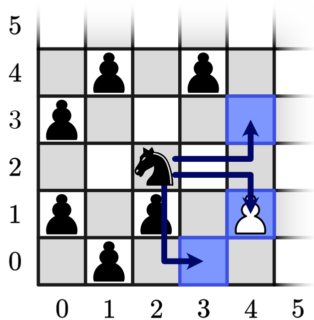


Le cavalier ne peut cependant pas sortir de l'échiquier, ce qui limite ses mouvements lorsqu'il est trop près du bord. Par exemple dans le cas suivant, le cavalier, trop proche du coin inférieur gauche de l'échiquier, n'aura que 4 mouvements autorisés.

1. La colonne est habituellement plutôt désignée par une lettre, mais il sera plus pratique pour nos programmes d'utiliser un entier à la place.



La présence ou l'absence d'autres pièces sur les cases au dessus desquelles passe le cavalier ne gêne pas son mouvement. En revanche, la case d'arrivée doit, soit être vide, soit contenir une pièce de la couleur opposée à celle du cavalier. Dans ce deuxième cas, on dit que la pièce en question est capturée par le cavalier à l'issue du mouvement.
Par exemple dans le cas suivant, le cavalier noir ♞ n'a que 3 mouvements valides, car 5 des 8 cases d'arrivée *a priori* possibles sont occupées par des pièces noires ♟. La présence d'une pièce sur la case $(2, 1)$ ne bloque en revanche pas son mouvement. L'un des 3 mouvements conduit à la capture d'une pièce blanche ♙ : celui qui atteint la case $(4, 1)$.



1.2 Graphes, chemins, flots

On entendra dans ce sujet par "graphe" un graphe orienté pondéré. Un tel graphe $\mathcal{G}$ est un triplet $\mathcal{G} = (\mathcal{S}, \mathcal{A}, \mathsf{poids})$. $\mathcal{S}$ est l'ensemble des sommets de $\mathcal{G}$. $\mathcal{A} \subseteq \mathcal{S}^2$ est celui de ses arcs : un arc $(u, v) \in \mathcal{A}$ est dit sortant de u et entrant dans v. On ajoute la restriction que $\mathcal{A}$ ne peut jamais contenir à la fois (u, v) et (v, u) : $\forall u, v \in \mathcal{S}.\ (u, v) \in \mathcal{A} \Rightarrow (v, u) \notin \mathcal{A}$.
Enfin $\mathsf{poids} : \mathcal{A} \mapsto \mathbb{Z}$ est une fonction attribuant à chaque arc un entier appelé son poids.
Pour un sommet $u \in \mathcal{S}$ on notera $\mathsf{succ}_{\mathcal{G}}(u)$ l'ensemble des successeurs de u dans $\mathcal{G}$:

$$\mathsf{succ}_{\mathcal{G}}(u) = \{v \in \mathcal{S} \mid (u, v) \in \mathcal{A}\}.$$

Un chemin C sur le graphe $\mathcal{G}$ est une séquence de $l + 1 \in \mathbb{N}$ sommets reliés par des arcs : $\mathsf{C} = [u_0, \dots, u_l]$, pour des $u_i \in \mathcal{S}$ tels que $\forall i \in [\![0, l-1]\!].\ (u_i, u_{i+1}) \in \mathcal{A}$. u_0 est le départ de C, u_l son arrivée. Le poids du chemin C est la somme des poids de ses arcs : $\mathsf{poids}(\mathsf{C}) = \sum_{0 \le i \le l-1} \mathsf{poids}(u_i, u_i + 1)$.

Considérons un graphe $\mathcal{G} = (\mathcal{S}, \mathcal{A}, \mathsf{poids})$, contenant deux sommets particuliers $s, t \in \mathcal{S}$, tels que s ne possède que des arcs sortants, et t que des arcs entrants, c'est-à-dire $\forall u \in \mathcal{S}.\ (u, s) \notin \mathcal{A}\ \wedge\ (t, u) \notin \mathcal{A}$. On appellera s une source et t un puits. Soit $f \in \mathbb{N}$ un entier. On appelle flot de valeur f de s vers t tout sous ensemble $\mathsf{F} \subseteq \mathcal{A}$ tel que

- F contient f arcs sortants de s : $\big|\{(x, y) \in \mathsf{F} \mid x = s\}\big| = f$
- F contient f arcs entrants dans t : $\big|\{(x, y) \in \mathsf{F} \mid y = t\}\big| = f$

— pour tout autre sommet $u \in \mathcal{S}$, F contient autant d'arcs entrants dans u que sortants de u :

$$\forall u \in \mathcal{S} \setminus \{s,t\}.\ \left|\{(x,y) \in \mathsf{F} \mid x = u\}\right| = \left|\{(x,y) \in \mathsf{F} \mid y = u\}\right|$$

On note alors $\mathsf{valeur}(\mathsf{F}) = f$. On appelle le poids du flot F la somme des poids de ses arcs :

$$\mathsf{poids}(\mathsf{F}) = \sum_{(u,v)\in\mathsf{F}} \mathsf{poids}(u,v)$$

2 Génération de données

Pour commencer, nous allons définir quelques suites générant des données pseudo-aléatoires qui serviront au test numérique des fonctions des parties suivantes.
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 $a = b \times q + r$.
De plus, pour une liste l et un élément x, on note $x::l$ la liste obtenue en ajoutant x au début de l. $\mathsf{longueur}(l)$ désigne le nombre d'éléments de l, et `[]` la liste vide. Par exemple, si l est la liste $[1,2,2]$ alors $3::l$ désigne une nouvelle liste $[3,1,2,2]$, et $\mathsf{longueur}(l) = 3$.

2.1 Génération de nombres pseudo-aléatoires

Étant donné u_0, on définit par récurrence :

$$\forall i \in \mathbb{N}.\ u_{i+1} = (989\,909 \times u_i) \bmod 1\,212\,149$$

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

a) $u_1 \bmod 1\,000$ ***b)*** $u_{255} \bmod 1\,000$ ***c)*** $u_{2\,023} \bmod 1\,000$

d) $u_{54\,321} \bmod 1\,000$

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.

```
(* u0~ utilisé dans le sujet *)
let u0 : int = 1530

(* mémoïsation : on précalcule toutes les valeurs jusqu'à max_n dans une table *)
let max_n : int = 10000000
let un_tab : int array = Array.make (max_n + 1) (-1)

let _ =
  un_tab.(0) <- u0;
  for i = 1 to max_n do
    un_tab.(i) <- (989909 * un_tab.(i-1)) mod 1212149
  done

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

let q1 (n:int) : int =
  (un n) mod 1000
```

2.2 Génération de positions sur un échiquier

On définit, pour tous $n, p, i \in \mathbb{N}$ tels que $n \neq 0$ le couple d'entiers dans $[\![0, n-1]\!]$ suivant :

$$v_{n,p}(i) = ((9\,029 \times u_{p+2\times i}) \text{ mod } n,\ (9\,029 \times u_{p+2\times i+1}) \text{ mod } n)$$

Pour tous $n, p \in \mathbb{N}$ tels que $n \neq 0$ et $p \leq n^2$, on construit une liste $\mathsf{Pos}_{n,p}$ de longueur p contenant les $v_{n,p}(i)$ pour $i = 1, 2, 3, \ldots$, en éliminant les doublons. Plus précisément, $\mathsf{Pos}_{n,p}$ est la valeur calculée par l'algorithme suivant :

```
P := []
i := 1
tant que longueur(P) < p
    si v_{n,p}(i) ∉ P alors
        P := v_{n,p}(i) :: P
    i := i + 1
renvoyer P
```

Algorithme 1 : Calcul de $\mathsf{Pos}_{n,p}$

$\mathsf{Pos}_{n,p}$ contient ainsi p couples deux à deux distincts d'entiers de $[\![0, n-1]\!]$, c'est-à-dire p positions distinctes sur un échiquier de taille $n \times n$.

Question 2 *Écrire un programme qui calcule* $\mathsf{Pos}_{n,p}$, *et donner, pour les valeurs de* n, p *suivantes, la somme* $\left(\sum_{(i,j)\in\mathsf{Pos}_{n,p}} i + j\right) \text{ mod } 1\,000$ *:*

a) $n = 8$, $p = 30$ **b)** $n = 120$, $p = 1000$ **c)** $n = 1\,300$, $p = 200\,000$

```
type pos = int * int

(* calcul de v_n,p *)
let vnp (n:int) (p:int) (i:int) : pos =
  let x = (un (p + 2*i)) * 9029 in
  let y = (un (p + 2*i + 1)) * 9029 in
  (x mod n, y mod n)

(* calcul de la liste Pos_n,p *)
let pos (n:int) (p:int) : pos list =
  (* test d'appartenance efficace, avec un tableau pour marquer les positions déjà
  ↪  vues *)
  let mem = Array.make_matrix n n false in
  let rec pos_aux (restant:int) (i:int) (a:pos list) : pos list =
    match restant with
    | 0 -> a
    | _ ->
      let v = vnp n p i in
      if mem.(fst v).(snd v) then
        pos_aux restant (i+1) a
      else
        (mem.(fst v).(snd v) <- true;
         pos_aux (restant-1) (i+1) (v::a))
  in
  pos_aux p 1 []

let q2 (n:int) (p:int) : int =
  let pp = pos n p in
  let s = List.fold_left (fun s (i,j) -> s + i + j) 0 pp in
  s mod 1000
```

Question à développer pendant l'oral 1 *Décrire le fonctionnement de votre programme : comment le test d'appartenance et l'ajout à la liste sont-ils implémentés ? Évaluer leur complexité en temps.*

Il faut faire attention à implémenter le test d'appartenance en temps $\mathcal{O}(1)$, donc sans parcourir à chaque fois la liste P. On peut pour cela tenir à jour une table de booléens de taille $n \times n$, indiquant pour chaque (i, j) si cette position a déjà été vue.
L'ajout à la liste doit se faire en temps $\mathcal{O}(1)$, c'est-à-dire qu'il ne faut pas générer une nouvelle copie de P à chaque fois.
On peut noter aussi que la longueur de P dans la condition doit être obtenue en temps $\mathcal{O}(1)$, donc sans parcourir P, par exemple en tenant à jour un compteur.
On obtient ainsi une complexité en temps linéaire dans le nombre de $v_{n,p}(\cdot)$ testés.

2.3 Génération d'échiquiers avec scores

Pour tous entiers naturels non nuls n, p, i, j, on définit

$$w_{n,p}(i, j) = ((7\,681 \times u_p + i) \bmod n) \times n + ((8\,059 \times u_p + j) \bmod n) + 1$$

Question 3 *Écrire une fonction qui calcule w, et en donner les valeurs suivantes* $\underline{\bmod 10\,000}$ *:*

a) $w_{12,20}(6, 11)$ ***b)*** $w_{340,1\,000}(268, 64)$ ***c)*** $w_{1\,300,200\,000}(653, 208)$

```
type score = int

let wnp (n:int) (p:int) (i:int) (j:int) : score =
  let u = un p in
  let x = 7681*u + i in
  let y = 8059*u + j in
  (x mod n) * n + (y mod n) + 1

let q3 (n:int) (p:int) (i:int) (j:int) : int =
  (wnp n p i j) mod 10000
```

On manipulera dans la suite du sujet des échiquiers $\mathcal{E}$ dont certaines cases contiendront des pièces associées à un entier non nul appelé score. On note $\mathsf{Pieces}(\mathcal{E})$ l'ensemble des coordonnées où sont placées des pièces sur $\mathcal{E}$. Pour une case $(i,j) \in \mathsf{Pieces}(\mathcal{E})$ contenant une pièce, $\mathsf{Score}_{\mathcal{E}}(i,j)$ désigne le score qui lui est associé. Pour une case $(i,j) \notin \mathsf{Pieces}(\mathcal{E})$ ne contenant pas de pièce, on considèrera que $\mathsf{Score}_{\mathcal{E}}(i,j) = 0$.

Pour tous $n,p,k \in \mathbb{N}$ tels que $n \neq 0$ et $p+k \leq n^2$, on notera $\mathcal{E}_{n,p,k}$ l'échiquier de taille $n \times n$ contenant $p+k$ pièces, aux coordonnées $\mathsf{Pos}_{n,p+k}$. Le score associé $\mathsf{Score}_{\mathcal{E}_{n,p,k}}(i,j)$ est l'entier positif $w_{n,p+k}(i,j)$ si (i,j) est l'un des p premiers éléments de $\mathsf{Pos}_{n,p+k}$, et l'entier négatif $-w_{n,p+k}(i,j)$ si c'est l'un des k derniers.

Question 4 *Écrire un programme qui calcule $\mathcal{E}_{n,p,k}$, et donner pour les valeurs suivantes de n,p,k la valeur absolue* $\left| \sum_{(i,j)\in \mathsf{Pieces}(\mathcal{E}_{n,p,k})} \mathsf{Score}_{\mathcal{E}_{n,p,k}}(i,j) \right| \bmod 10\,000.$

a) *n=8, p=23, k=10* ***b)*** *n=174, p=1 700, k=1 000* ***c)*** *n=3 400, p=4 300, k=5 000*

```
(* représentation de l'échiquier par liste des pièces + tableau des scores *)
type echiquier = {mutable pcs:pos list; tab: (score array array)}
let cree n = {pcs=[]; tab=Array.make_matrix n n 0}
let pieces e = e.pcs
let score e (i,j) = e.tab.(i).(j)
let ajoute e (i,j) s = e.pcs <- (i,j)::e.pcs; e.tab.(i).(j) <- s

(* calcul de E_n,p,k *)
let enpk (n:int) (p:int) (k:int) : echiquier =
  let e = cree n in
  let positions = ref (pos n (p+k)) in
  for x=1 to p do
    let (i, j) = List.hd !positions in
    positions := List.tl !positions;
    ajoute e (i,j) (wnp n (p+k) i j)
  done;
  for x=1 to k do
    let (i, j) = List.hd !positions in
    positions := List.tl !positions;
    ajoute e (i,j) (- (wnp n (p+k) i j))
  done;
  e

let q4 (n:int) (p:int) (k:int) : int =
  let e = enpk n p k in
  let s = List.fold_left (fun s c -> s + (score e c)) 0 (pieces e) in
  (abs s) mod 10000
```

Question à développer pendant l'oral 2 *Décrire la structure de données que vous avez employée pour représenter $\mathcal{E}$. Évaluer pour cette structure de données la complexité en temps des opérations consistant à obtenir le score d'une position (i, j), et à parcourir les positions de* $\mathsf{Pieces}(\mathcal{E})$.

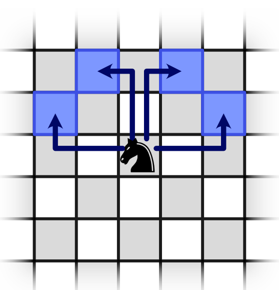
La question indique qu'il faut faire attention au temps nécessaire pour obtenir le score d'une position, et pour parcourir les pièces – on voit dans la suite que ces opérations seront souvent utilisées.
Idéalement, on choisit donc une structure de données qui permet l'accès au score en temps $\mathcal{O}(1)$, et l'itération sur les pièces en temps $\mathcal{O}(p)$. Par exemple, on peut garder la paire d'une table $n \times n$ donnant le score de chaque position, et de la liste $\mathsf{Pieces}(\mathcal{E})$. Alternativement, une table de hachage/dictionnaire est efficace aussi.

3 Marche d'un cavalier

Dans cette partie, on s'intéresse à un échiquier $\mathcal{E}$ de taille $n \times n$, avec $n \in \mathbb{N}^\star$, sur lequel sont placées p pièces blanches, sur p cases distinctes (avec $p \in [\![0, n^2]\!]$).

Du point de vue d'un joueur, il peut être plus utile de capturer certaines pièces, stratégiquement placées, que d'autres. Pour modéliser ceci, on supposera donc qu'à chacune des p pièces blanches est associé un score positif distinct, qui représentera l'intérêt de capturer cette pièce.

Un cavalier noir ♞ se déplace sur l'échiquier. On se restreint au cas où le cavalier commence dans la case $(0, 0)$ et ne se déplace jamais vers le bas. Il a donc à chaque coup au plus 4 mouvements possibles, représentés ci-dessous.



Pour une position (i, j) sur un échiquier de taille $n \times n$, on appelle $\mathsf{mouv}_n(i, j)$ la liste des (au plus 4) positions sur l'échiquier qu'un cavalier peut atteindre en un coup depuis (i, j) avec cette restriction.

Question 5 *Écrire une fonction qui calcule* mouv. *Pour les valeurs suivantes de n, p, i, si l'on note* $\mathsf{mouv}_n(v_{n,p}(i)) = [(i_1, j_1), \ldots, (i_k, j_k)]$, *donner la valeur* $\left(\sum_{1 \le m \le k} (i_m^2 + j_m) \right) \bmod 1\,000$.

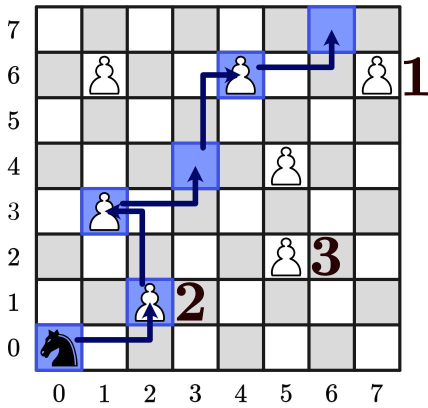
a) $n = 8, p = 10, i = 6$ ***b)*** $n = 120, p = 148, i = 54$ ***c)*** $n = 600, p = 2\,530, i = 567$

```
let dans_l_echiquier (n:int) (i,j:pos) : bool =
  0 <= i && i < n && 0 <= j && j < n

let mouv (n:int) (i,j:pos) : pos list =
  let ps = [(i-2, j+1); (i-1, j+2); (i+1, j+2); (i+2, j+1)] in
  List.filter (dans_l_echiquier n) ps

let q5 (n:int) (p:int) (i:int) : int =
  let mv = mouv n (vnp n p i) in
  let s = List.fold_left (fun s (i,j) -> s + i*i + j) 0 mv in
  s mod 1000
```

On appellera marche dans l'échiquier $\mathcal{E}$ une succession de mouvements faisables à partir de $(0,0)$, c'est-à-dire toute séquence $\mathsf{M} = [(i_0, j_0), \ldots, (i_k, j_k)]$ de positions de $\mathcal{E}$ telle que $\forall m \in [\![0, k-1]\!]$. $(i_{m+1}, j_{m+1}) \in \mathsf{mouv}_n(i_m, j_m)$, et $i_0 = j_0 = 0$.
Le score de la marche M dans l'échiquier $\mathcal{E}$, $\mathsf{Score}_{\mathcal{E}}(\mathsf{M})$, désigne la somme des scores de toutes les cases parcourues :

$$\mathsf{Score}_{\mathcal{E}}(\mathsf{M}) = \sum_{1 \leq m \leq k} \mathsf{Score}_{\mathcal{E}}(i_m, j_m).$$

Notons que le score éventuel de la case de départ n'est pas compté.
La figure suivante donne un exemple d'une marche sur un échiquier $\mathcal{E}$ de taille 8×8 contenant 7 pièces dont les scores sont indiqués à côté des cases correspondantes (le score nul des cases vides n'est pas représenté).



Cette marche est $\mathsf{M} = [(0,0), (2,1), (1,3), (3,4), (4,6), (6,7)]$, et son score est $\mathsf{Score}_{\mathcal{E}}(\mathsf{M}) = 2 + 4 + 9 = 15$.

On s'intéresse à un cavalier qui cherche à maximiser le score de sa marche. On se pose le problème suivant : étant donné un échiquier $\mathcal{E}$ de taille $n \times n$, calculer un tableau des scores TS de taille $n \times n$ tel que pour tous i, j, $\mathsf{TS}[i][j]$ contient le score maximal des marches se terminant en (i, j) dans $\mathcal{E}$ (ou 0, s'il n'existe aucune telle marche).

3.1 Recherche exhaustive

On se propose tout d'abord de résoudre ce problème en considérant une par une toutes les marches. On initialise la table TS de façon que toutes ses cases contiennent 0. On énumère ensuite successivement toutes les marches partant de $(0,0)$. Pour chaque marche M se terminant sur une position (i,j), si $\mathsf{Score}_{\mathcal{E}}(\mathsf{M})$ est plus grand que la valeur courante de $\mathsf{TS}[i][j]$, on met à jour TS.

Question 6 *Calculer la table* TS *en énumérant exhaustivement toutes les marches, pour les échiquiers* $\mathcal{E}_{n,p,0}$ *avec les* n, p *suivants. Pour chacun, donner la somme* $\left(\sum_{0\leq i\leq n-1,\, 0\leq j\leq n-1} \mathsf{TS}[i][j]\right) \bmod 10\,000$.

a) $n = 8, p = 17$ ***b)*** $n = 15, p = 175$ ***c)*** $n = 16, p = 190$

```
let tscores (n:int) (e:echiquier) : score array array =
  let t = Array.make_matrix n n 0 in
  let rec aux (s:score) (i,j:pos) =
    if t.(i).(j) < s then
      t.(i).(j) <- s;
    let mv = mouv n (i,j) in
    List.iter
      (fun c -> aux (s + (score e c)) c)
      mv
  in
  aux 0 (0,0);
  t

let q6 (n:int) (p:int) : int =
  let e = enpk n p 0 in
  let t = tscores n e in
  let s = Array.fold_left (Array.fold_left (fun s x -> s + x)) 0 t in
  s mod 10000
```

Question à développer pendant l'oral 3 *Décrire le fonctionnement de votre programme pour le calcul exhaustif de* TS*. Évaluer sa complexité en temps.*

L'algorithme proposé revient à faire un parcours en profondeur de l'arbre dans lequel chaque position (i, j) a pour fils les positions accessibles depuis elle en un coup avec les mouvements autorisés. Il fait à chaque appel des comparaisons et affectations en $\mathcal{O}(1)$, puis $|\mathsf{mouv}_n(i, j)| \leq 4$ appels récursifs. La profondeur de la pile d'appels est bornée par la longueur des marches. Comme le cavalier monte d'au moins une rangée à chaque coup, la longueur d'une marche est au plus n. L'algorithme a donc une complexité en temps en $\mathcal{O}(4^n)$.

3.2 Algorithme de Bellman-Ford

La méthode de recherche exhaustive n'est pas très performante. Pour calculer la table TS plus efficacement, on se propose à présent d'utiliser l'algorithme de Bellman-Ford.

Il s'agit d'un algorithme permettant de calculer des chemins de poids maximal dans un graphe. On décrit ici son fonctionnement de façon générale sur un graphe quelconque, et il vous faudra ensuite déterminer comment l'appliquer au problème qui nous intéresse.

Considérons un graphe $\mathcal{G} = (\mathcal{S}, \mathcal{A}, \mathsf{poids})$ et un sommet $s \in \mathcal{S}$. L'algorithme calcule une table PC, telle que pour tout sommet $u \in \mathcal{S}$, $\mathsf{PC}[u]$ est le poids maximal d'un chemin de s à u dans $\mathcal{G}$, s'il en existe, ou une valeur par défaut $\perp$, s'il n'en existe pas.

L'algorithme est le suivant :

```
Initialiser PC à ⊥ : pour chaque u ∈ S  PC[u] := ⊥
PC[s] := 0
répéter jusqu'à ce que PC cesse de changer, ou au plus |S| fois
    pour chaque u ∈ S tel que PC[u] ≠ ⊥
        pour chaque v ∈ succ_G(u)
            si PC[v] = ⊥ ou PC[u] + poids(u, v) > PC[v] alors
                PC[v] := PC[u] + poids(u, v)
            sinon
                ne rien faire
si on a atteint |S| tours et PC a encore changé au tour |S| alors
    renvoyer erreur
sinon
    renvoyer PC
```

Algorithme 2 : Algorithme de Bellman-Ford

La boucle principale est exécutée au plus $|\mathcal{S}|$ fois. Si durant un tour de boucle PC ne change pas, alors il ne changera plus jamais : on s'arrête donc et on renvoie PC. Si on a atteint $|\mathcal{S}|$ tours sans que PC ne se stabilise, l'algorithme renvoie une erreur.

Remarque. Si l'algorithme renvoie erreur, cela signifie qu'il n'existe pas de chemin de poids maximal. En effet, on peut montrer que $\mathcal{G}$ contient alors un chemin revenant d'un sommet u à lui-même dont le poids soit strictement positif – un cycle positif. Un tel cycle peut être emprunté plusieurs fois pour produire des chemins de poids arbitrairement grands.

On ne demande pas de justifier que les graphes auxquels vous appliquerez l'algorithme par la suite ne contiennent pas de tels cycles.

Question 7 *En utilisant l'algorithme de Bellman-Ford, calculer la table* TS *des scores maximaux des marches depuis* $(0,0)$*, pour les échiquiers* $\mathcal{E}_{n,p,0}$ *avec les* n, p *suivants. Pour chacun, donner la valeur*

$$\left(\sum_{0\le i\le n-1,\, 0\le j\le n-1} \mathsf{TS}[i][j]\right) \bmod 10\,000.$$

a) $n = 8, p = 20$ ***b)*** $n = 50, p = 1\,600$ ***c)*** $n = 100, p = 2\,540$

```ocaml
let bellmanford (n:int) (e:echiquier) : (score option) array array =
  let pc = Array.make_matrix n n None in
  pc.(0).(0) <- Some 0;
  let changement = ref true in
  let tours = ref 0 in
  while !changement && !tours < n * n do
    changement := false;
    tours := !tours + 1;
    for i=0 to n-1 do
      for j=0 to n-1 do
        let mv = mouv n (i,j) in
        List.iter
          (fun (i',j') ->
             match pc.(i).(j), pc.(i').(j') with
             | None, _ -> ()
             | Some p, None ->
               changement := true;
               pc.(i').(j') <- Some (p + (score e (i', j')))
             | Some p, Some p' when p' < p + (score e (i', j')) ->
               changement := true;
               pc.(i').(j') <- Some (p + (score e (i', j')))
             | _ -> ())
          mv;
      done;
    done;
  done;
  if !changement && !tours = n * n then
    failwith "ERREUR : cycle de poids positif";
  pc

let q7 (n:int) (p:int) : int =
  let e = enpk n p 0 in
  let pc = bellmanford n e in
  let s =
    Array.fold_left
      (Array.fold_left
         (fun s x -> match x with
            | None -> s
            | Some x -> s + x))
      0 pc in
  s mod 10000
```

Question à développer pendant l'oral 4 *Expliquer de quelle façon vous avez modélisé le problème posé sur l'échiquier en un problème de graphes, pour pouvoir lui appliquer l'algorithme de Bellman-Ford.*

Pour un échiquier $\mathcal{E}$, on construit le graphe $\mathcal{G}_\mathcal{E} = (\mathcal{S}_\mathcal{E}, \mathcal{A}_\mathcal{E}, \mathsf{poids}_\mathcal{E})$ où

— $\mathcal{S}_\mathcal{E} = [\![0, n-1]\!]^2$;

— $\mathcal{A}_\mathcal{E} = \{(u, v) \mid v \in \mathsf{mouv}_n(u)\}$;

— $\mathsf{poids}_\mathcal{E}(u, v) = \mathsf{Score}_\mathcal{E}(v)$.

Il existe une bijection évidente entre les chemins partant de $(0, 0)$ dans $\mathcal{G}_\mathcal{E}$ et les marches dans $\mathcal{E}$, qui associe à un chemin C une marche M se terminant sur la même position, et telle que $\mathsf{Score}_\mathcal{E}(\mathsf{M}) = \mathsf{poids}_\mathcal{E}(\mathsf{C})$. Le score maximal des marches jusqu'à (i, j) dans $\mathcal{E}$ est donc égal au poids maximal des chemins de $(0, 0)$ à (i, j) dans $\mathcal{G}_\mathcal{E}$, que l'on calcule avec l'algorithme de Bellman-Ford.

Ce n'était pas demandé, mais on peut noter dans ce corrigé que le graphe ainsi construit ne contient pas de cycle de poids positif.

Question à développer pendant l'oral 5 *Évaluer la complexité en temps de votre programme.*

Avec le graphe proposé, on a $|\mathcal{S}_{\mathcal{E}}| = n^2$. Le programme implémentant l'algorithme de Bellman-Ford fait donc au plus n^2 tours de la boucle principale. À chaque tour, il parcourt les n^2 sommets. Pour chaque sommet u, il parcourt ses $|\mathsf{succ}_{\mathcal{G}_{\mathcal{E}}}(u)| \leq 4$ successeurs. Pour chacun il effectue des accès aux scores dans $\mathcal{E}$ et des comparaisons, c'est-à-dire des opérations en $\mathcal{O}(1)$. En tout, on peut donc majorer sa complexité par $\mathcal{O}(n^4)$.
On peut même remarquer que le nombre de tours de boucles avant que la table PC se stabilise est en fait majoré par la longueur maximale d'un chemin dans le graphe. En effet, après k tours de boucle, $\mathsf{PC}[u]$ contient le poids maximal des chemins jusqu'à u d'au plus k arcs. PC ne change donc plus une fois la longueur maximale d'un chemin dépassée. Ici, on a déjà observé que les chemins sont de longueur au plus n : on fait donc en fait au plus n tours de boucle, ce qui donne une meilleure majoration, en $\mathcal{O}(n^3)$.

L'algorithme de Bellman-Ford peut être étendu pour permettre de calculer, non seulement le poids maximal des chemins depuis s, mais aussi un chemin atteignant ce poids maximal.

On construit, en plus de la table PC, une table Pred : pour un sommet u, $\mathsf{Pred}[u]$ désignera un sommet qui précède u dans un chemin de poids maximal de s à u. En utilisant Pred, on peut à l'issue de l'algorithme calculer un chemin de poids maximal vers chaque sommet.

On munit pour cette question les positions de l'ordre suivant : $(i_1, j_1) < (i_2, j_2)$ si et seulement si $i_1 < i_2$ ou ($i_1 = i_2$ et $j_1 < j_2$). On munit alors les marches d'un ordre, appelé ordre lexicographique inverse, défini comme suit : $\mathsf{M} < \mathsf{M}'$ lorsqu'il existe des entiers a, b, c et des positions $(m_i)_i$, $(m'_i)_i$, $(p_i)_i$ tels que $\mathsf{M} = [m_0, \ldots, m_a, p_0, \ldots, p_b]$, $\mathsf{M}' = [m'_0, \ldots, m'_c, p_0, \ldots, p_b]$, et $m_a < m'_c$.

Question 8 *Implémenter cette extension de l'algorithme de Bellman-Ford, pour calculer la marche* M *de* $(0,0)$ *à* $(n-1, n-1)$ *la plus petite pour l'ordre lexicographique inverse parmi celles de score maximal, dans l'échiquier* $\mathcal{E}_{n,p,0}$*, pour les valeurs de* n, p *suivantes. Donner, à chaque fois, la valeur* $\left(\sum_{(i,j)\in\mathsf{M}} i^2 + j\right) \bmod 10\,000$.

a) $n = 8, p = 20$ ***b)*** $n = 50, p = 1\,600$ ***c)*** $n = 100, p = 2\,540$

```ocaml
(* fonction de la question précédente étendue avec la table des prédécesseurs *)
let bellmanford_ext (n:int) (e:echiquier) : pos option array array =
  let pc = Array.make_matrix n n None in
  let pred = Array.make_matrix n n None in
  pc.(0).(0) <- Some 0;
  let changement = ref true in
  let tours = ref 0 in
  while !changement && !tours < n * n do
    changement := false;
    tours := !tours + 1;
    for i=0 to n-1 do
      for j=0 to n-1 do
        let mv = mouv n (i,j) in
        List.iter
          (fun (i',j') ->
             match pc.(i).(j), pc.(i').(j') with
             | None, _ -> ()
             | Some p, None ->
               changement := true;
               pc.(i').(j') <- Some (p + (score e (i', j')));
               pred.(i').(j') <- Some (i,j)
             | Some p, Some p' when p' < p + (score e (i', j')) ->
               changement := true;
               pc.(i').(j') <- Some (p + (score e (i', j')));
               pred.(i').(j') <- Some (i,j)
             | Some p, Some p' when p' = p + (score e (i', j')) ->
               (match pred.(i').(j') with
                | None -> assert false;
                | Some (i'', j'') when i < i'' || (i = i'' && j < j'') ->
                  changement := true; (* attention : pred a changé, même si pc non
                  ↪  *)
                  pred.(i').(j') <- Some (i,j)
                | _ -> () )
             | _ -> ())
          mv;
      done;
    done;
  done;
  if !changement && !tours = n * n then
    failwith "ERREUR : cycle de poids positif";
  pred

(* reconstruction du chemin à partir de la table des prédécesseurs *)
let chemin_bellmanford (pred:pos option array array) (p:pos) : pos list =
  let x = ref p in
  let c = ref [] in
  while !x <> (0,0) do
    c := !x :: !c;
    match pred.(fst !x).(snd !x) with
    | None -> failwith "ERREUR : pas de chemin";
    | Some y -> x := y
  done;
  c := (0,0) :: !c;
  !c
```

```
let q8 (n:int) (p:int) : int =
  let e = enpk n p 0 in
  let pred = bellmanford_ext n e in
  let c = chemin_bellmanford pred (n-1, n-1) in
  let s = List.fold_left (fun s (i,j) -> s + i*i + j) 0 c in
  s mod 10000
```

Question à développer pendant l'oral 6 *Décrire la façon dont votre programme calcule la table des prédécesseurs et la marche de score maximal la plus petite.*

Comme on le voit dans le programme proposé, il faut faire attention dans l'algorithme de Bellman-Ford ainsi étendu à modifier la condition d'arrêt : il faut que non seulement PC mais aussi Pred soient stabilisés pour s'arrêter. On doit aussi prendre garde à ne modifier le sommet stocké dans Pred$[u]$ que si l'on trouve un autre prédécesseur possible qui soit plus petit pour l'ordre lexicographique.
De cette façon, on calcule le chemin voulu, le plus petit pour l'ordre lexicographique inverse, en remontant à l'issue de l'algorithme de Bellman-Ford la table des prédécesseurs, depuis $(n-1, n-1)$ jusque $(0, 0)$. Il peut *a priori* arriver qu'aucun tel chemin n'existe : on peut alors renvoyer une erreur. Ce cas ne se produisait jamais avec les valeurs demandées.

3.3 Calcul efficace

Cette question est indépendante de la suite du sujet.

Question 9 *Écrire un programme implémentant un algorithme plus efficace pour le calcul du score maximal d'une marche de* $(0, 0)$ *à* $(n-1, n-1)$ *sur l'échiquier* $\mathcal{E}_{n,p,0}$*. Donner ce score maximal* $\underline{\bmod 10\,000}$ *pour les valeurs de* n*,* p *suivantes.*

a) $n = 2\,000, p = 1\,000$ **b)** $n = 2\,500, p = 20\,000$ **c)** $n = 3\,000, p = 100\,000$

```ocaml
(* les positions depuis lesquelles on peut atteindre (i,j) *)
let mouv_inv (n:int) (i,j:pos) : pos list =
  let ps = [(i-2, j-1); (i-1, j-2); (i+1, j-2); (i+2, j-1)] in
  List.filter (dans_l_echiquier n) ps

(* Algorithme dynamique, O(n^2) en espace.
   Peut être rendu O(n) en ne gardant que la ligne courante et les 2 précédentes. *)
let dyn (n:int) (e:echiquier) : score =
  let ts = Array.make_matrix n n None in
  ts.(0).(0) <- Some 0;
  for j=0 to n-1 do
    for i=0 to n-1 do
      List.iter
        (fun (i', j') ->
           match ts.(i').(j'), ts.(i).(j) with
           | Some p', None -> ts.(i).(j) <- Some (p' + (score e (i, j)))
           | Some p', Some p when p' + (score e (i, j)) > p -> ts.(i).(j) <- Some
           ↪   (p' + (score e (i, j)))
           | _ -> ())
        (mouv_inv n (i, j));
    done;
  done;
  match ts.(n-1).(n-1) with
  | Some s -> s
  | None -> failwith "ERREUR : pas de chemin de poids max"
```

```ocaml
let q9 (n:int) (p:int) : int =
  let e = enpk n p 0 in
  let s = dyn n e in
  s mod 10000
```

Question à développer pendant l'oral 7 *Décrire le fonctionnement de votre algorithme, et évaluer sa complexité en temps et en espace.*

La façon naturelle de procéder est par programmation dynamique. On peut remarquer que le score maximal pour une case (i, j) se calcule à partir de (au plus) 4 cases qui permettent de l'atteindre, qui se trouvent toutes une ou deux lignes en dessous. Si l'on note, comme à la question 6, TS la table telle que $\mathsf{TS}[i, j]$ contient le score maximal d'une marche jusque (i, j), ou $\perp$ si aucune telle marche n'existe, on a :

$$\mathsf{TS}[i,j] = \begin{cases} \mathsf{Score}_{\mathcal{E}}(i,j) + \max\big(\{\mathsf{TS}[u] \mid u \in \{(i-2,j-1),(i-1,j-2),(i+1,j-2),(i+2,j-1)\} \\ \qquad\qquad \wedge\, \mathsf{TS}[u] \neq \perp \,\wedge\, u \text{ est dans l'échiquier}\}\big) \quad \text{si de tels } u \text{ existent} \\ \perp \quad \text{sinon.} \end{cases}$$

On peut donc remplir la table TS ligne par ligne, en commençant par la ligne $j = 0$, initialisée avec $\mathsf{TS}[0, 0] = 0$ et $\perp$ partout ailleurs.
On a alors une complexité en temps en $\mathcal{O}(n^2)$, puisqu'on fait un seul parcours de TS avec $\mathcal{O}(1)$ opérations pour chaque case.
La complexité en espace est en $\mathcal{O}(n^2)$, pour stocker TS. On peut l'améliorer en $\mathcal{O}(n)$, en ne stockant que la ligne courante et les deux précédentes, comme c'est habituel pour les algorithmes dynamiques.

4 Charge de cavalerie

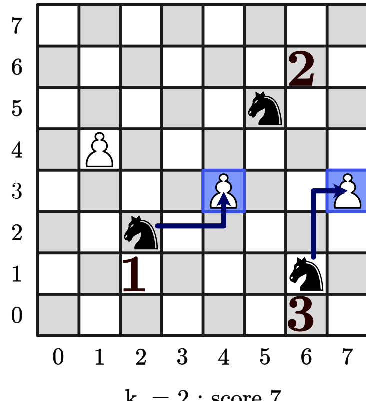
On s'intéresse dans cette dernière partie à la question de coordonner l'attaque de multiples cavaliers pour maximiser le score obtenu.

Toujours sur un échiquier $\mathcal{E}$ de taille $n \times n$, on considère de nouveau p pièces blanches, sur des cases distinctes, avec chacune un score associé. On ajoute en plus k cavaliers noirs sur d'autres cases, qui ont eux aussi chacun un score associé. Ceci suppose bien sûr que $p + k \leq n^2$. Pour distinguer plus facilement les pièces blanches des cavaliers, on suppose que les scores des pièces blanches sont positifs, et ceux des cavaliers négatifs.

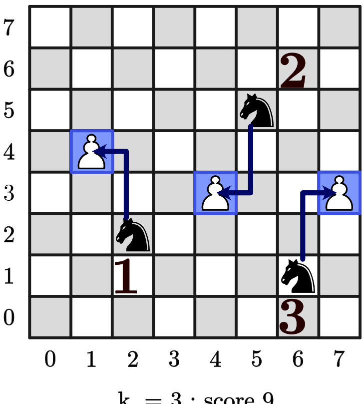
Contrairement à la partie précédente, on ne restreint plus le mouvement des cavaliers à aller vers le haut : ils ont donc de nouveau jusqu'à 8 déplacements possibles à chaque coup. On considèrera que capturer une pièce blanche de score v avec un cavalier de score v' rapporte un score $v + v'$.

Le problème que l'on va chercher à résoudre est le suivant : étant donné un tel $\mathcal{E}$, et un entier naturel $k' \leq k$, quel est le score maximal que l'on peut atteindre en faisant capturer une pièce en un coup chacun, à exactement k' cavaliers ? Deux cavaliers différents ne pouvant bien sûr pas capturer la même pièce.

Par exemple, pour un échiquier 8×8, avec $p = 3$ pièces et $k = 3$ cavaliers avec les positions et les valeurs de la figure suivante, on peut atteindre un score de 7 en jouant $k' = 2$ cavaliers, et 9 en jouant $k' = 3$ cavaliers.



k = 2 : score 7



k = 3 : score 9

4.1 Algorithme glouton

On souhaite tenter de résoudre ce problème en adoptant une stratégie gloutonne, consistant à choisir à chaque fois le coup le plus rentable parmi les coups disponibles, c'est-à-dire celui rapportant le score le plus élevé, jusqu'à avoir joué k' coups (on renverra une erreur si aucun coup n'est disponible alors que l'on n'a pas encore joué k' coups).

Question 10 *Écrire un programme qui implémente cette approche gloutonne. Donner* la valeur absolue, mod 10 000, *du score maximal atteignable sur les échiquiers $\mathcal{E}_{n,p,k}$ avec les nombres de captures k' suivants.*

a) $\mathcal{E}_{8,20,20}, k' = 10$

b) $\mathcal{E}_{600,40\,000,40\,000}$, $k' = 10\,000$

c) $\mathcal{E}_{1\,000,120\,000,98\,000}$, $k' = 20\,000$

```ocaml
(* positions des pièces capturables par la pièce en (i,j) dans e *)
let capturables (n:int) (e:echiquier) (i,j:pos) : pos list =
  if score e (i, j) >= 0 then [] (* pas de pièce en (i,j) *)
  else
    let cap =
      [(i-2, j-1); (i-2, j+1); (i-1, j-2); (i-1, j+2);
       (i+1, j-2); (i+1, j+2); (i+2, j-1); (i+2, j+1)]
    in
    let cap = List.filter (dans_l_echiquier n) cap in
    List.filter (fun c -> score e c > 0) cap

(* positions des pièces pouvant capturer la pièce en (i,j) dans e *)
let captureurs (n:int) (e:echiquier) (i,j:pos) : pos list =
  if score e (i,j) <= 0 then [] (* pas de pièce en (i,j) *)
  else
    let cap =
      [(i-2, j-1); (i-2, j+1); (i-1, j-2); (i-1, j+2);
       (i+1, j-2); (i+1, j+2); (i+2, j-1); (i+2, j+1)]
    in
    let cap = List.filter (dans_l_echiquier n) cap in
    List.filter (fun c -> score e c < 0) cap
```

```
let glouton (n:int) (e:echiquier) (k':int) : score =
  let l = pieces e in
  let coups_possibles =
    List.concat_map
      (fun p ->
         List.map
           (fun x -> (p,x))
           (capturables n e p))
      l
  in
  let cmp (c1, p1:pos*pos) (c2, p2:pos*pos) : int =
    (score e p2) + (score e c2) - (score e p1) - (score e c1)
  in
  (* liste des coups possibles triée par rentabilité décroissante *)
  let coups_possibles = List.sort cmp coups_possibles in

  (* table des cavaliers et pièces déjà utilisés. un tableau convient aussi. *)
  let utilises = Hashtbl.create (List.length l) in

  let rec aux s k' coups_possibles =
    match k', coups_possibles with
    | 0, _ -> s
    | _, [] ->
      failwith "ERREUR : échec à capturer k' pièces"
    | _, (pos_c, pos_p) :: cc
      when Hashtbl.mem utilises pos_c || Hashtbl.mem utilises pos_p ->
      aux s k' cc
    | _, (pos_c, pos_p) :: cc ->
      Hashtbl.add utilises pos_c true;
      Hashtbl.add utilises pos_p true;

      let s' = s + (score e pos_p) + (score e pos_c) in
      aux s' (k'-1) cc
  in
  aux 0 k' coups_possibles

let q10 (n:int) (p:int) (k:int) (k':int) : int =
  let e = enpk n p k in
  let s = glouton n e k' in
  (abs s) mod 10000
```

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

On pourrait envisager un algorithme glouton naïf, qui va à k' reprises considérer tous les coups restants possibles, choisir l'un de ceux qui rapportent le meilleur score, enlever le cavalier et la pièce utilisés, et recommencer. Il faut pour cela à chaque fois parcourir la liste des $\mathcal{O}(p+k)$ pièces et cavaliers restants, et on aurait ainsi une complexité en $\mathcal{O}(k' \cdot (p+k))$. C'est trop inefficace pour traiter en temps raisonnable les valeurs demandées.
L'algorithme proposé est donc légèrement plus malin : on commence par calculer une fois pour toutes la liste de tous les coups possibles, ce qui se fait en temps $\mathcal{O}(p+k)$. On trie ensuite cette liste par score décroissant, ce qui se fait en temps $\mathcal{O}((p+k) \cdot \log(p+k))$. On tient également à jour une table (ici une table de hachage, mais un tableau convient aussi) indiquant quelles pièces ont déjà été utilisées. Il reste alors à parcourir la liste triée des coups possibles : on considère son premier élément, si les deux pièces impliquées sont toujours disponibles on le sélectionne et on les marque comme non disponibles, sinon on

l'élimine et on continue. On parcourt potentiellement toute la liste, c'est-à-dire qu'on considère $\mathcal{O}(p+k)$ coups, et pour chacun on fait $\mathcal{O}(1)$ opérations.
On obtient ainsi une complexité en $\mathcal{O}((p+k)\cdot\log(p+k))$.

Question à développer pendant l'oral 9 *L'algorithme glouton renvoie-t-il toujours le score maximal que l'on souhaitait calculer ? Expliquer.*

L'algorithme glouton ne renvoie pas toujours le score maximal. La figure en début de section 4 en donne en fait un contre-exemple : le glouton choisit bien les coups optimaux pour $k'=2$, mais si $k'=3$ il est ensuite bloqué et n'a pas de troisième coup disponible, alors qu'il était possible de capturer trois pièces en jouant d'autres premier et deuxième coup.

4.2 Algorithme de Ford-Fulkerson

Pour finir, on se propose de résoudre le problème en passant de nouveau par l'algorithmique des graphes. Pour cela, on verra le problème de capture des pièces par les cavaliers comme une instance du problème du flot de poids maximal (*cf.* Partie 1.2 pour la définition d'un flot).
Ce problème est le suivant : étant donné un graphe $\mathcal{G}$ avec une source s et un puits t, et $f\in\mathbb{N}$, quel est le poids maximal d'un flot de valeur f de s à t dans $\mathcal{G}$ (s'il en existe) ?

Le problème de capture qui nous intéresse peut être vu comme une instance de ce problème. Pour un échiquier $\mathcal{E}$ de taille $n\times n$ contenant p pièces et k cavaliers, on considère le graphe $\mathcal{G}(\mathcal{E})=(\mathcal{S},\mathcal{A},\mathsf{poids})$ suivant :

— Ses $k+p+2$ sommets sont les positions des cavaliers et pièces sur $\mathcal{E}$, plus une source s et un puits t.

$$\mathcal{S}=\mathsf{Pieces}(\mathcal{E})\cup\{s,t\}$$

La valeur utilisée pour s et t importe peu, il faut simplement qu'on ne puisse pas les confondre avec les autres sommets.

— Des arcs relient chaque cavalier aux pièces qu'il peut capturer, la source s à chaque cavalier, et chaque pièce au puits t :

$$\begin{aligned}\mathcal{A}=&\left\{((i,j),(i',j'))\in(\mathcal{S}\setminus\{s,t\})^2\;\middle|\;\mathsf{Score}_{\mathcal{E}}(i,j)<0\wedge\mathsf{Score}_{\mathcal{E}}(i',j')>0\wedge(i',j')\in\mathsf{captures}(i,j)\right\}\\&\cup\left\{(s,(i,j))\;\middle|\;(i,j)\in\mathcal{S}\setminus\{s,t\}\;\wedge\;\mathsf{Score}_{\mathcal{E}}(i,j)<0\right\}\\&\cup\left\{((i,j),t)\;\middle|\;(i,j)\in\mathcal{S}\setminus\{s,t\}\;\wedge\;\mathsf{Score}_{\mathcal{E}}(i,j)>0\right\}\end{aligned}$$

où $\mathsf{captures}(i,j)$ désigne l'ensemble des positions qu'un cavalier en (i,j) peut capturer en un coup.

— les arcs reliant s aux cavaliers et les pièces à t ont un poids nul, l'arc reliant un cavalier à une pièce a pour poids le score correspondant :

$$\forall(u,v)\in\mathcal{A}.\;\mathsf{poids}(u,v)=\begin{cases}\mathsf{Score}_{\mathcal{E}}(i,j)+\mathsf{Score}_{\mathcal{E}}(i',j') & \text{si } u=(i,j)\neq s \text{ et } v=(i',j')\neq t\\ 0 & \text{si } u=s \text{ ou } v=t\end{cases}$$

Intuitivement, cet encodage fonctionne de la façon suivante : trouver un flot de valeur f de s à t dans $\mathcal{G}(\mathcal{E})$ donne un ensemble d'arcs, dont f sortent de s, et entrent donc dans f cavaliers. De chacun de ces f cavaliers sort alors un arc vers une pièce, sélectionnant ainsi la pièce qu'il doit capturer. Le poids de f est égal au score obtenu en capturant ces f pièces.
On admettra que le score maximal atteignable sur $\mathcal{E}$ en k' captures, que l'on recherche, est égal au poids maximal d'un flot de valeur k' de s à t dans $\mathcal{G}(\mathcal{E})$.

On utilisera, pour le calcul numérique des réponses aux questions 11 et 12, l'ensemble $\mathsf{F}_0(\mathcal{E}) = \{((i,j),(i',j')) \in \mathcal{A} \mid \{(i,j),(i',j')\} \cap \{s,t\} = \emptyset \ \wedge\ i' \bmod 2 = 0\}$ des arcs ne reliant ni s ni t qui entrent dans un sommet d'abscisse paire. $\mathsf{F}_0(\mathcal{E})$ est un sous-ensemble des arcs de $\mathcal{G}(\mathcal{E})$. Il ne s'agit pas en général d'un flot, mais ce sera sans importance pour les calculs demandés.

L'algorithme de Ford-Fulkerson permet de calculer un flot de poids maximal. Il repose sur la construction d'un graphe résiduel. Étant donné $\mathcal{G} = (\mathcal{S}, \mathcal{A}, \mathsf{poids})$ et un flot F, le graphe résiduel de $\mathcal{G}$ pour F, $\mathcal{R}_\mathsf{F}(\mathcal{G}) = (\mathcal{S}, \mathcal{A}_{\mathcal{R}_\mathsf{F}(\mathcal{G})}, \mathsf{poids}_{\mathcal{R}_\mathsf{F}(\mathcal{G})})$, possède les mêmes sommets que $\mathcal{G}$. Ses arcs sont ceux de $\mathcal{G}$, à ceci près que l'on inverse le sens et le poids de ceux qui sont présents dans F. C'est-à-dire que $(u,v) \in \mathcal{A}_{\mathcal{R}_\mathsf{F}(\mathcal{G})}$ si et seulement si

- $(u,v) \in \mathcal{A} \setminus \mathsf{F}$, et alors $\mathsf{poids}_{\mathcal{R}_\mathsf{F}(\mathcal{G})}(u,v) = \mathsf{poids}(u,v)$;
- ou bien $(v,u) \in \mathsf{F}$ et alors $\mathsf{poids}_{\mathcal{R}_\mathsf{F}(\mathcal{G})}(u,v) = -\mathsf{poids}(v,u)$.

Remarque. Cette construction est bien définie, car on a imposé en 1.2 que $\mathcal{A}$ ne contienne jamais à la fois (u,v) et (v,u).

Question 11 *Écrire un programme qui, étant donné $\mathcal{E}$ et F, calcule le graphe $\mathcal{R}_\mathsf{F}(\mathcal{G}(\mathcal{E}))$. Pour les $\mathcal{E}$ suivants, avec $\mathsf{F} = \mathsf{F}_0(\mathcal{E})$, donner la valeur absolue*

$$\left| \sum_{u \in \mathcal{S}_{\mathcal{R}_\mathsf{F}(\mathcal{G}(\mathcal{E}))}} \sum_{v \in \mathsf{succ}_{\mathcal{R}_\mathsf{F}(\mathcal{G}(\mathcal{E}))}} \mathsf{poids}_{\mathcal{R}_\mathsf{F}(\mathcal{G}(\mathcal{E}))}(u,v) \right| \bmod 10\,000.$$

a) $\mathcal{E}_{8,20,20}$ ***b)*** $\mathcal{E}_{60,250,140}$ ***c)*** $\mathcal{E}_{110,850,400}$

```
type flot = (pos * pos) list

(* sommets du graphe résiduel *)
let rf_sommets (n:int) (e:echiquier) (f:flot) : pos list =
  let s = (n, 0) in
  let t = (0, n) in
  s :: t :: (pieces e)

(* arcs du graphe résiduel *)
let rf_succ (n:int) (e:echiquier) (f:flot) (p:pos) : pos list =
  let s = (n,0) in
  let t = (0,n) in
  let l = pieces e in
  if p = s then (* succ s : les cavaliers p' tels que (s, p') n'est pas dans f *)
    List.filter (fun p' -> (score e p') < 0 && not (List.mem (s, p') f)) l
  else if p = t then (* succ t : les pieces p' telles que (p', t) est dans f *)
    List.filter (fun p' -> (score e p') > 0 && (List.mem (p', t) f)) l
  else if score e p < 0 then (* succ cavalier *)
    let c = capturables n e p in (* les pieces p' à sa portée tq (p, p') pas dans f
    ↪ *)
    let l = List.filter (fun p' -> not (List.mem (p,p') f)) c in
    let l = if List.mem (s,p) f then s::l else l in (* + s si (s, p) est dans f *)
    l
  else (* succ pièce *)
    let c = captureurs n e p in (* les cavaliers p' à sa portée tq (p', p) est dans
    ↪  f *)
    let l = List.filter (fun p' -> List.mem (p',p) f) c in
    let l = if List.mem (p,t) f then l else t::l in (* + t si (p,t) pas dans f *)
    l
```

```
(* poids des arcs du graphe résiduel *)
let rf_poids (n:int) (e:echiquier) (f:flot) (p:pos) (p':pos) : score =
  let s = (n,0) in
  let t = (0,n) in
  if p = s || p = t || p' = s || p' = t then 0
  else if score e p < 0 && score e p' > 0 && not (List.mem (p,p') f) then
    (score e p) + (score e p')
  else if score e p > 0 && score e p' < 0 && (List.mem (p',p) f) then
    - ((score e p) + (score e p'))
  else 0

(* pseudo-flot F_n *)
let fn (n:int) (e:echiquier) : flot =
  let l =
    List.concat_map
      (fun p ->
         List.map
           (fun x -> (p,x))
           (capturables n e p))
      (pieces e) in
  List.filter (fun (p,p') -> (fst p' mod 2 = 0)) l

let  q11 (n:int) (p:int) (k:int) : int =
  let e = enpk n p k in
  let f = fn n e in
  let s =
    List.fold_left
      (fun s p ->
         List.fold_left
           (fun s p' ->
              s + rf_poids n e f p p')
           s
           (rf_succ n e f p))
      0
      (rf_sommets n e f)
  in
  (abs s) mod 10000
```

En utilisant cette notion de graphe résiduel, nous pouvons maintenant présenter l'algorithme de Ford-Fulkerson pour le calcul du poids maximal d'un flot de valeur f dans $\mathcal{G} = (\mathcal{S}, \mathcal{A}, \mathsf{poids})$.

```
F := []
tant que valeur(F) < f
    C := chemin de s à t de poids maximal dans R_F(G)
    si un tel C existe alors
        mettre à jour F comme suit :
            pour chaque arc (u, v) de C
                si (u, v) ∈ A alors ajouter (u, v) à F
                si (v, u) ∈ A alors retirer (v, u) de F
    sinon
        renvoyer erreur
renvoyer poids(F)
```

Algorithme 3 : Algorithme de Ford-Fulkerson

F est à chaque étape un flot, initialement vide, et que l'on va remplir progressivement jusqu'à atteindre la valeur f cherchée. À chaque étape, on cherchera un chemin qui permettra d'augmenter la valeur de F. Si c'est impossible, alors on admet qu'il n'existe aucun flot de valeur f, et on renverra une erreur.
Le calcul du chemin de poids maximal dans $\mathcal{R}_{\mathsf{F}}(\mathcal{G})$ peut être réalisé avec l'algorithme de Bellman-Ford, vu dans la partie 3.2.

Question 12 *Écrire un programme qui, étant donné $\mathcal{E}$, F, calcule un chemin de poids maximal de s à t dans $\mathcal{R}_{\mathsf{F}}(\mathcal{G}(\mathcal{E}))$ avec l'algorithme de Bellman-Ford. Donner le poids de ce chemin pour les $\mathcal{E}$ suivants, avec* $\mathsf{F} = \mathsf{F}_0(\mathcal{E})$.

a) $\mathcal{E}_{8,20,20}$ ***b)*** $\mathcal{E}_{60,250,140}$ ***c)*** $\mathcal{E}_{110,850,400}$

Indication. On pourra bien sûr réutiliser le programme écrit pour la question 8, en prenant soin de l'adapter au graphe utilisé ici. On admettra que $\mathcal{R}_{\mathsf{F}}(\mathcal{G}(\mathcal{E}))$ ne contient pas de cycle positif – c'est le cas au cours de l'algorithme de Ford-Fulkerson.

```
(* version de la fonction de Q8 adaptée pour utiliser le graphe résiduel *)
let bellmanford_ext_rf (n:int) (e:echiquier) (f:flot) : pos option array array =
  let sommets = rf_sommets n e f in
  let nsommets = List.length sommets in

  let pc = Array.make_matrix (n+1) (n+1) None in (* n+1 pour avoir la place de
  ↪ stocker s,t *)
  let pred = Array.make_matrix (n+1) (n+1) None in (* on perd un peu d'espace mais
  ↪ tant pis *)
  pc.(n).(0) <- Some 0;

  let changement = ref true in
  let tours = ref 0 in

  while !changement && !tours < nsommets do
    tours := !tours + 1;
    changement := false;
    List.iter
      (fun p ->
         let (i,j) = p in
         let succs = rf_succ n e f p in
         List.iter
           (fun p' ->
              let (i',j') = p' in
              let poids_pp' = rf_poids n e f p p' in
              match pc.(i).(j), pc.(i').(j') with
              | None, _ -> ()
              | Some po, None ->
                changement := true;
                pc.(i').(j') <- Some (po + poids_pp');
                pred.(i').(j') <- Some p
              | Some po, Some po' when po' < po + poids_pp' ->
                changement := true;
                pc.(i').(j') <- Some (po + poids_pp');
                pred.(i').(j') <- Some p
              | _ -> ())
           succs)
      sommets;
  done;
  if !changement && !tours = nsommets then
    failwith "ERREUR : cycle de poids positif";
  pred
```

```
(* reconstruction du chemin à partir de la table des prédécesseurs *)
let chemin_bellmanford_rf (n:int) (pred:pos option array array) : pos list =
  let s = (n,0) in
  let t = (0,n) in
  let x = ref t in
  let c = ref [] in
  while !x <> s do
    c := !x :: !c;
    match pred.(fst !x).(snd !x) with
    | None -> failwith "ERREUR : pas de chemin";
    | Some y -> x := y
  done;
  c := s :: !c;
  !c

let q12 (n:int) (p:int) (k:int) : int =
  let e = enpk n p k in
  let f = fn n e in
  let pred = bellmanford_ext_rf n e f in
  let c = chemin_bellmanford_rf n pred in
  let s, _ =
    List.fold_left
      (fun (s,p) p' -> (s + (rf_poids n e f p p'), p'))
      (0, List.hd c)
      (List.tl c)
  in
  s
```

Question 13 *Écrire un programme qui, étant donné $\mathcal{E}$ et k', calcule le score maximal atteignable en capturant k' pièces blanches par k' cavaliers. On utilisera l'algorithme de Ford-Fulkerson pour calculer le poids maximal d'un flot de valeur k' sur le graphe $\mathcal{G}(\mathcal{E})$ décrit précédemment. Donner ce score maximal pour les entrées suivantes.*

a) $\mathcal{E}_{8,31,31}, k' = 25$ ***b)*** $\mathcal{E}_{10,47,45}, k' = 38$ ***c)*** $\mathcal{E}_{20,150,150}, k' = 120$

```
let fordfulkerson (n:int) (e:echiquier) (k':int) : score =
  let s = (n,0) in
  let t = (0,n) in
  let f = ref [] in
  (* tant que f n'a pas k' arcs sortant de s *)
  while (List.length (List.filter (fun (p,p') -> p = s) !f)) < k' do
    (* calcul d'un chemin augmentant dans R_F *)
    let pred = bellmanford_ext_rf n e !f in
    let c = chemin_bellmanford_rf n pred in

    (* mise à jour de f :
       pour chaque arc a de c
       - si a est dans G : ajouter a à f (il n'y est forcément pas encore)
       - sinon -a est dans G : retirer -a de f (il y est forcément déjà) *)
    let ajoutes = ref [] in
    let retires = ref [] in
    let p = ref (List.hd c) in
```

```
List.iter
      (fun p' ->
         (* arcs de G : s -> cavalier, cavalier -> pièce, ou pièce -> t *)
         if ((!p = s && p' <> t && score e p' < 0)
             || (!p <> s && p' <> t && score e !p < 0 && score e p' > 0)
             || (p' = t && !p <> s && score e !p > 0))
         then ajoutes := (!p, p') :: !ajoutes
         else retires := (p', !p) :: !retires;
         p := p')
      (List.tl c);

    f := !ajoutes @ (List.filter (fun x -> not (List.mem x !retires)) !f);
  done;
  let s = List.fold_left (fun s (p, p') -> s + rf_poids n e [] p p') 0 !f in
  s

let q13 (n:int) (p:int) (k:int) (k':int) : int =
  let e = enpk n p k in
  fordfulkerson n e k'
```

Question à développer pendant l'oral 10 *Évaluer la complexité en temps de votre programme.*

Il faut tout d'abord déterminer la complexité de l'algorithme de Bellman-Ford appliqué sur $\mathcal{R}_{\mathsf{F}}(\mathcal{G}(\mathcal{E}))$ au cours de la boucle principale du programme. On pourrait être tenté de réutiliser la complexité calculée à la question d'oral 5, mais il faut être prudent : cette analyse supposait que l'on dispose en temps constant de la description du graphe. Ici, $\mathcal{R}_{\mathsf{F}}(\mathcal{G}(\mathcal{E}))$ est calculé, et l'on doit donc prendre en compte ce temps de calcul. Il dépend bien sûr de la représentation choisie pour ce graphe et pour le flot F. On le détaille ici pour la solution proposée.
Pour un sommet u, le calcul de $\mathsf{succ}_{\mathcal{R}_{\mathsf{F}}(\mathcal{G}(\mathcal{E}))}$ demande de parcourir F un nombre constant de fois, et se fait donc en $\mathcal{O}(p+k)$, puisque F est représenté par une liste d'arcs de taille $\mathcal{O}(p+k)$. Le calcul de $\mathsf{poids}_{\mathcal{R}_{\mathsf{F}}(\mathcal{G}(\mathcal{E}))}(u)$ est en $\mathcal{O}(p+k)$, pour la même raison. Le graphe résiduel $\mathcal{R}_{\mathsf{F}}(\mathcal{G}(\mathcal{E}))$ contient de plus $\mathcal{O}(p+k)$ sommets. En replaçant les complexités des calculs des successeurs et poids dans l'analyse de complexité de l'algorithme de Bellman-Ford, on obtient une complexité en $\mathcal{O}((p+k)^3)$.

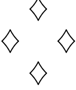
Dans l'algorithme de Ford-Fulkerson, la valeur $\mathsf{valeur}(\mathsf{F})$ du flot courant F est initialement nulle, et augmente de 1 exactement à chaque tour de la boucle principale. On en effectue donc k' tours, puisque la valeur à atteindre pour s'arrêter est k'. À chaque tour :

— L'algorithme de Bellman-Ford est appelé sur $\mathcal{R}_{\mathsf{F}}(\mathcal{G}(\mathcal{E}))$: $\mathcal{O}((p+k)^3)$ opérations ;

— Le chemin est calculé à partir de la table des prédécesseurs renvoyée : $\mathcal{O}(p+k)$ opérations, puisque ce chemin est de longueur au plus $p+k$;

— Les $\mathcal{O}(p+k)$ arcs de ce chemin sont ajoutés ou retirés de F : avec la représentation choisie, cela demande un parcours du chemin (avec des opérations en $\mathcal{O}(1)$ pour chaque élément), puis un parcours de F avec une opération en $\mathcal{O}(p+k)$ (test d'appartenance au chemin) pour chaque élément. La mise à jour de F requiert donc $\mathcal{O}((p+k)^2)$ opérations. C'est bien sûr améliorable, mais cette complexité est de toute manière absorbée par celle de l'algorithme de Bellman-Ford.

Au total, on obtient une complexité temporelle en $\mathcal{O}(k' \cdot (p+k)^3)$. On pourrait faire mieux avec une meilleure structure de données pour F, et mieux encore en calculant le graphe résiduel $\mathcal{R}_{\mathsf{F}}$ de façon incrémentale au lieu de le recalculer à chaque tour de boucle. Ces optimisations n'étaient cependant pas nécessaires pour répondre à la question.



Fiche réponse type : Charge de cavalerie

$\widetilde{u_0}$: 1530

Question 1

a) 669

b) 392

c) 224

d) 391

Question 2

a) 216

b) 789

c) 201

Question 3

a) 39

b) 503

c) 5056

Question 4

a) 411

b) 5390

c) 1573

Question 5

a) 92

b) 752

c) 716

Question 6

a) 1980

b) 6127

c) 6372

Question 7

a) 2334

b) 7362

c) 9893

Question 8

a) 115

b) 2873

c) 7666

Question 9

a) 474

b) 7317

c) 922

Question 10

a) 234

b) 8575

c) 8539

Question 11

a) 254

b) 4024

c) 5314

Question 12

a) 62

b) 121

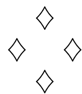
c) 11992

Question 13

a) 215

b) -2

c) 6772



Des marches dans des graphes

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

Durée de l'épreuve : 3 heures 30 minutes

Juin/Juillet 2023

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.

1 Préliminaires

1.1 Notations

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

Ce sujet portant sur des graphes pondérés orientés, chaque utilisation du mot graphe doit être entendue comme graphe pondéré orienté. Un graphe est donc, pour ce sujet, un triplet $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathsf{poids})$ où $\mathcal{N}$ est l'ensemble fini des sommets ou nœuds du graphe ; $\mathcal{E} \subset \mathcal{N}^2$ est l'ensemble des arcs, chaque arc étant une paire de sommets ; et $\mathsf{poids} : \mathcal{N} \to \mathbb{N}$ est une fonction associant à chaque sommet du graphe un poids positif. Notez que cette définition interdit les arcs multiples (deux sommets ne peuvent être reliés que zéro ou une fois), mais qu'il est possible d'avoir un arc d'un sommet s à lui-même, par l'arc (s, s). Un arc (s, t) de $\mathcal{E}$ est appelé un arc sortant de s, et le sommet t est un successeur de s. On notera $\mathsf{succ}_{\mathcal{G}}(s)$ l'ensemble des successeurs de s dans $\mathcal{G}$:

$$\mathsf{succ}_{\mathcal{G}}(s) = \{t \mid (s, t) \in \mathcal{E}\}$$

Par souci de simplicité, dans ce sujet, $\mathcal{N}$ est toujours constitué des nombres de 0 à $n - 1$ avec n le nombre de sommets.

1.2 Génération de nombres pseudo-aléatoires

Étant donné u_0, on définit la récurrence :

$$\forall t \in \mathbb{N}, \quad u_{t+1} = (909\,091 \times u_t) \mod 1\,010\,101\,039$$

L'entier u_0 vous est donné, et doit être recopié sur votre fiche réponse avec vos résultats. Une fiche réponse type vous est donnée en exemple, et contient tous les résultats attendus pour une valeur de u_0 différente de la vôtre (notée $\widetilde{u_0}$). Il vous est conseillé de tester vos algorithmes avec cet $\widetilde{u_0}$ et de comparer avec la fiche de résultats fournie. Pour chaque calcul demandé, avec le bon choix d'algorithme le calcul ne devrait demander qu'au plus de l'ordre de quelques secondes, jamais plus d'une minute.

On aura souvent besoin de nombreuses valeurs consécutives de la suite u_n. Il est donc conseillé que votre implémentation calcule le tableaux de tous les u_n jusqu'à un certain rang.

Question 1 *Calculer les valeurs suivantes :*

- ***a)*** $u_1 \mod 1000$
- ***b)*** $u_{12} \mod 1000$
- ***c)*** $u_{1\,234} \mod 1000$
- ***d)*** $u_{7\,654\,321} \mod 1000$

```
let next_u (x : int) : int = 909_091 * x mod 1_010_101_039

(* Tableau des n premiers éléments de la suite, dans l'ordre
   [array_un = \[u_0, ..., u_{n-1}\]] *)
let array_un (n : int) : int array =
  let a = Array.make n 0 in

  let rec fill (i : int) (ui : int) =
    if i = n then ()
    else begin a.(i) <- ui; fill (i + 1) (next_u ui) end
  in

  fill 0 u0;
  a
```

1.3 Extraction d'une sous-liste croissante

Soit L une liste $(a_1, \ldots, a_n)$ d'entiers positifs. On note incr-list(L) la sous-liste de L obtenue en ne gardant un a_i que si il est plus grand que tous les éléments précédents dans la liste. Par exemple, incr-list$(1, 4, 2, 3, 6, 5) = (1, 4, 6)$, puisque 2 et 3 ont été exclus car plus petits que 4 (qui les précède), et de même 5 a été exclu car plus petit que 6.
Formellement, si $L = (a_1, \ldots, a_n)$ alors incr-list$(L) = (b_1, \ldots, b_k)$ telle que $(b_1, \ldots, b_k)$ est une liste d'entiers strictement croissante ($b_i < b_j$ quand $i < j$) et :

$$\{b_1, \ldots, b_k\} = \{a_i \mid \forall j < i,\, a_j < a_i\}$$

Question 2 *Soit v_n la suite telle que $\forall n \in \mathbb{N}$, $v_n = u_n \mod 9$. On pose L_n la liste $(v_0, \ldots, v_{n-1})$. Implémenter la fonction incr-list$(\cdot)$, et utiliser votre implémentation pour calculer la somme des entiers de incr-list(L_n), c'est à dire $\left(\sum_{s \in \text{incr-list}(L_n)} s\right)$ pour les valeurs de n suivants :*

a) $n = 3$ ***b)*** $n = 6$ ***c)*** $n = 123$ ***d)*** $n = 1\,234$

Pour tester votre code, vous pouvez vous aider du fait que pour $\widetilde{u_0}$ on a $L_6 = (5, 5, 4, 3, 7, 2)$ et donc incr-list$(L_6) = (5, 7)$.

```
let extract_sublist (l : int list) : int list =
  let rec extr (max : int) (l : int list) (acc : int list) =
    match l with
    | [] -> List.rev acc
    | h :: t ->
      if h > max then extr h t (h :: acc)
      else extr max t acc
  in
  extr (-1) l []

(* build list `L_n` *)
let ln : int -> int list =
  let un = array_un 7654322 in
  fun n -> List.init n (fun i -> un.(i) mod 9)
```

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

Question (très) simple pour commencer.
Soit n la longueur de la liste. Une implémentation naïve garde un élément u_i en testant tous les u_j pour $j < i$, et est donc en $\mathcal{O}(n^2)$. On peut faire mieux, en parcourant la liste de gauche à droite, et en gardant à tout instant le plus grand élément déjà rencontré, ce qui donne une implémentation en $\mathcal{O}(n)$.

1.4 Génération de graphe

Étant donné trois entiers strictement positifs n, M et p, on note $G(n, M, p)$ le graphe $(\mathcal{N}, \mathcal{E}, \mathsf{poids})$ où $\mathcal{N} = \{0, \ldots, n-1\}$ et pour chaque sommet $s \in \mathcal{N}$:

— La liste des arcs sortants potentiels du sommet s est la liste :

$$(s, t_0), \ldots, (s, t_{M-1}) \qquad \text{où } \forall 0 \leq i < M,\ t_i = u_{s \times M + i} \mod n.$$

La liste des arcs sortants de s est obtenue en ne gardant dans $(s, t_0), \ldots, (s, t_{M-1})$ que les arcs vers des sommets croissants. Plus précisément,

$$\mathsf{succ}_{\mathcal{G}}(s) = \big\{t_j \mid t_j \in \mathsf{incr\text{-}list}(t_0, \ldots, t_{M-1})\big\}.$$

On remarquera que tous les sommets ont au plus M successeurs. De plus, puisque (s, t_0) est toujours un arc sortant de s, tous les sommets ont au moins 1 successeur.

— Le poids $\mathsf{poids}(s)$ du sommet s est $(u_{n \times M + s} \mod p)$.

Un exemple de graphe est donné Figure 1.

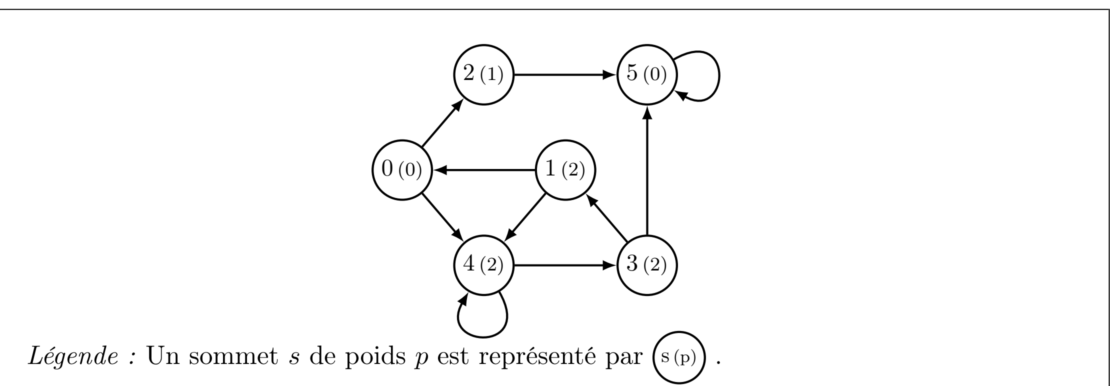


FIGURE 1 – Le graphe $G(6, 3, 3)$ pour $\widetilde{u_0}$.

On appelle degré sortant d'un sommet s le nombre de successeurs de s, c'est à dire $|\mathsf{succ}_{\mathcal{G}}(s)|$.

Question 3 *Implémenter une fonction calculant le graphe $G(\cdot, \cdot, \cdot)$, et l'utiliser pour calculer la somme $\sum_{s \in \mathcal{N}} |\mathsf{succ}_{\mathcal{G}}(s)|$ des degrés sortants des sommets des graphes $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathsf{poids})$ suivants :*

a) $G(6, 3, 10)$ ***b)*** $G(123, 7, 10)$ ***c)*** $G(1\,234, 10, 10)$ ***d)*** $G(10\,001, 22, 10)$

```
(* Un graphe dirigé [(n,d,poids)] :
   - [n] est le nombre de sommets, numérotés de [0] à [n-1].
   - [d] est le tableau des listes de transitions pour chaque sommet
     c-à-d que si [d.(i) = \[x1,...,xq\]] alors on a les transitions
     [i → x1], ..., [i → xq]
   - [p(i)] est le poids du [i]-ème sommet *)
type graph = int * int list Array.t * (int -> int)

let nodes (g : graph) : int = let n, _, _ = g in n

(* graphe à [n] sommets, [m] arcs, pour le paramètre [p] *)
let graph ~(n : int) ~(m : int) ~(p : int) : graph =
  let ln = array_un (n * (m + 1)) in

  let e =
    Array.init n
      (fun i ->
         List.init m (fun j -> ln.( i * m + j) mod n) |>
         extract_sublist
      )
  in
  (* poids, avec un tableau pré-calculé *)
  let p : int -> int =
    let w = Array.init n (fun i -> ln.( (n * m + i)) mod p) in
    fun i -> w.(i)
  in
  (n, e, p)

let sum_degrees (g : graph) : int =
  let _,e,_ = g in
  Array.fold_left (fun sum l -> sum + List.length l) 0 e
```

Question à développer pendant l'oral 2 *Décrire la structure de données que vous avez choisie pour représenter les graphes.*
On exprimera l'ordre de grandeur de l'espace mémoire utilisé par votre représentation en fonction du nombre de sommets n et du nombre d'arcs m des graphes.

On représente le graphe comme un tableau de n cases, où la i^e case contient la liste des arcs sortants de i. Il y a au plus $n \times M$ arcs dans le graphe, donc cette représentation est en $\mathcal{O}(n \times M)$. La représentation des poids est un simple tableau de n entiers, et est donc en $\mathcal{O}(n)$. Au total, on a donc une représentation en $\mathcal{O}(n \times M)$.
Si on note m le nombre d'arcs, notre représentation est en $\mathcal{O}(n + m)$.
Utiliser des listes d'adjacence est préférable ici, car M est toujours beaucoup plus petit que n dans ce sujet, et qu'on n'a jamais besoin de vérifier rapidement si un arc (i, j) est présent dans le graphe.

À partir de maintenant, on ne comptera pas le temps de calcul des graphes, ni leur espace mémoire, dans les évaluations de complexités d'algorithmes.

Hachage d'un graphe On définit une fonction de hachage sur les graphes comme suit : pour tout graphe $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathsf{poids})$,

$$\mathsf{hash}(\mathcal{G}) = \Big(\sum_{s\in\mathcal{N}} \sum_{t\in\mathsf{succ}_{\mathcal{G}}(s)} (s+1)\times(t+\mathsf{poids}(t)^2)\Big) \mod 1000.$$

Question 4 *Calculer la valeur de hash($\mathcal{G}$) pour les graphes $\mathcal{G}$ suivants :*

a) $G(6,3,10)$ ***b)*** $G(123,7,10)$ ***c)*** $G(1\,234,10,10)$ ***d)*** $G(10\,001,22,10)$

```
let rec ( *^ ) : int -> int -> int =
  fun i j -> if j = 0 then 1 else i * (i *^ (j - 1))

let hash_graph (g : graph) : int =
  let n,e,p = g in
  let sum = ref 0 in
  Array.iteri (fun i l ->
      sum :=
        List.fold_left (fun sum j ->
            sum + (i + 1) * (j + (p j) *^ 2)
          ) !sum l
    ) e;

  !sum mod 1000
```

2 Marches sur les graphes

Une stratégie sans mémoire π pour un graphe $(\mathcal{N}, \mathcal{E}, \mathsf{poids})$ est une fonction associant à chaque sommet s l'un de ses successeurs :

$$\pi : \mathcal{N} \to \mathcal{N} \quad \text{telle que} \quad \forall s \in \mathcal{N},\ \ (s, \pi(s)) \in \mathcal{E}.$$

Notez qu'il n'existe pas de telle fonction si le graphe possède un sommet sans successeur. Comme les sommets des graphes de ce sujet ont tous au moins un successeur, cela ne sera pas un problème.

Étant donné un graphe $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathsf{poids})$ avec $\mathcal{N} = \{0, \dots, n-1\}$, on note $\pi_0(\mathcal{G})$ la stratégie sans mémoire choisissant dans chaque sommet son plus petit successeur :

$$\forall s \in \mathcal{N},\ \ \pi_0(\mathcal{G})(s) = \min\{t \in \mathcal{N} \mid (s,t) \in \mathcal{E}\}.$$

2.1 Marches simples

Étant donné un graphe $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathsf{poids})$, une marche (ou chemin) ρ sur $\mathcal{G}$ est une liste de sommets $s_0, \dots, s_L$ reliés par des arcs, c'est à dire tel que $(s_i, s_{i+1}) \in \mathcal{E}$ pour tout $0 \le i < L$. On appelle L la longueur de la marche ρ : il s'agit du nombre d'arcs dans ρ.

Étant donné un sommet initial s et un entier L, une stratégie π définit une marche de longueur L en partant du sommet s et en se déplaçant dans le graphe selon la stratégie π. Plus précisément, on note $\mathsf{marche}(\mathcal{G}, L, \pi, s)$ la marche $s_0, \dots, s_L$ telle que $s_0 = s$ et $s_{i+1} = \pi(s_i)$ pour tout $0 \le i < L$.

Question 5 *Implémenter une fonction calculant la marche* marche$(\cdot,\cdot,\cdot,\cdot)$.
Pour un graphe $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \textit{poids})$ *où* $\mathcal{N} = \{0, \ldots, n-1\}$, *évaluer la somme modulo* 1000 *des sommets de la marche de longueur* L *au départ du sommet* $n-1$ *dans* $\mathcal{G}$ *suivant la stratégie* $\pi_0(\mathcal{G})$. *C'est-à-dire que si* marche$(\mathcal{G}, L, \pi_0(\mathcal{G}), n-1) = s_0, \ldots, s_L$, *alors on cherche la quantité :*

$$\left(\sum_{0\leq i\leq L} s_i\right) \ mod \ 1000$$

pour les graphes $\mathcal{G}$ *et longueurs* L *suivants :*

- ***a)*** $G(6, 3, 10), L = 4$
- ***b)*** $G(123, 7, 10), L = 20$
- ***c)*** $G(1\,234, 10, 10), L = 100$
- ***d)*** $G(10\,001, 22, 10), L = 5\,000$

```
type strat = int -> int

(* dans le sommet [i], la stratégie [pi0 i] choisit toujours
   le sommet de degré minimal dans le voisinage de [i] *)
let pi0 (g : graph) : strat =
  let n,d,_ = g in
  let strat =
    Array.init n
      (fun i ->
         List.fold_left (fun m j -> min m j) (List.hd d.(i)) d.(i)
      )
  in
  fun i -> strat.(i)

(* une marche *)
type run = int list

(* marche à l'envers de longueur [l] suivant [strat] dans [g] au
   départ de [s0].  [s0 → ... → sl] où [forall 0 <= i < l, s_{i+1} = strat
   s_i] *)
let run (strat : strat) (g : graph) (s0 : int) (l : int) : run =
  let rec exec (s0 : int) (l : int) acc =
    if l = 0 then (s0 :: acc) else exec (strat s0) (l - 1) (s0 :: acc)
  in
  exec s0 l []

let control_run (g : graph) ~(l : int) : int =
  let init = nodes g - 1 in
  let strat = pi0 g in
  let run = run strat g init l in
  ( List.fold_left (+) 0 run ) mod 1_000
```

Question à développer pendant l'oral 3 *Donner la complexité en temps et en mémoire de votre algorithme, en fonction du nombre de sommets* n, *du nombre d'arcs* m, *et de la longueur de la marche* L.

Le temps de calcul de $\pi_0(\mathcal{G})$ est en $\mathcal{O}(n \times M)$ ou $\mathcal{O}(n + m)$.
Le temps de calcul de la marche est en $\mathcal{O}(L)$.
La représentation mémoire de $\pi_0(\mathcal{G})$ est en $\mathcal{O}(n)$, et de la marche en $\mathcal{O}(L)$. Si on calcule le poids de la marche à la volée, alors on a un coût mémoire en $\mathcal{O}(1)$ (auquel il faut ajouter la représentation mémoire de $\pi_0(\mathcal{G})$).

2.2 Marches de concert avec seuil

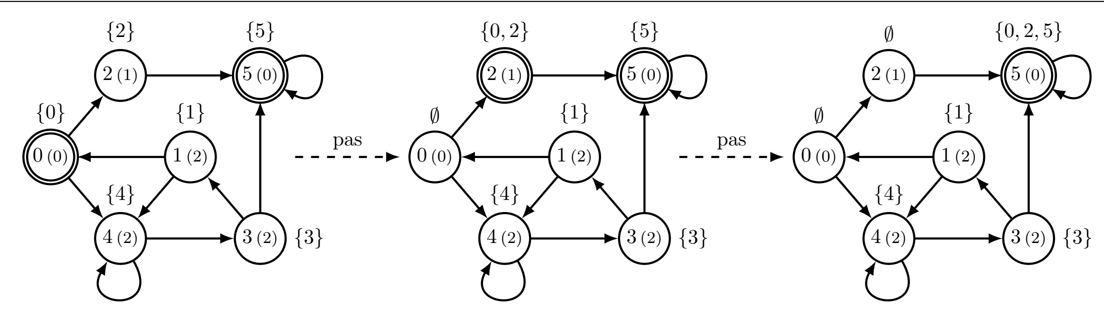
Cette section est indépendante de la section 3.

Soit un graphe $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathsf{poids})$ et une stratégie π de $\mathcal{G}$. Dans cette section, nous allons étudier des marches de concert dans le graphe : à chaque instant, un certain nombre d'agents seront répartis sur les sommets du graphe, et ceux-ci se déplaceront de concert et simultanément sur le graphe en suivant la même stratégie. De plus, ces déplacements seront soumis à un effet de seuil : pour qu'un groupe d'agents présents dans un sommet se déplace, il sera nécessaire qu'ils soient strictement plus nombreux que le poids du sommet.

Plus précisément, on considère un ensemble fini d'agents $\mathcal{A} = \{0, \ldots, n-1\}$ (il y a donc autant d'agents que de sommets). Ces agents sont répartis sur les sommets du graphe $\mathcal{G}$: il est possible qu'il y ait plusieurs agents sur le même sommet, et qu'il y ait des sommets sans agents. Initialement, chaque sommet du graphe contient exactement un agent : l'agent i est dans le sommet i. Tous les agents se déplacent sur le graphe en suivant la stratégie $\pi_0(\mathcal{G})$. Les agents essaient de se déplacer par groupe (tous les agents dans le sommet s se déplacent ensemble vers le sommet $\pi_0(\mathcal{G})(s)$), et les déplacements sont simultanés (tous les groupes d'agents se déplacent en même temps). Cependant, ces déplacements ne peuvent pas toujours avoir lieu : pour qu'un groupe d'agents $a_1, \ldots, a_k$ dans un sommet s puisse se déplacer en $\pi_0(\mathcal{G})(s)$, il est nécessaire qu'ils soient strictement plus nombreux que le poids du sommet s, c'est à dire que $k > \mathsf{poids}(s)$; si $k \leq \mathsf{poids}(s)$, les agents $a_1, \ldots, a_k$ restent dans le sommet s, en attendant que suffisamment d'autres agents les rejoignent. Une étape de déplacement de *tous* les groupes d'agents le pouvant est appelé un pas, et une séquence de pas décrit une marche de concert.
On souhaite effectuer une marche de concert pour un grand nombre de pas, et sur des graphes relativement importants. Pour être plus efficace, nous utiliserons la notion de sommet vivant : un sommet vivant est un sommet dans lequel il y a au moins un agent, et plus d'agents que le poids du sommet. Autrement dit, les sommets vivants sont ceux depuis lesquels les groupes d'agents vont se déplacer au prochain pas. La Figure 2 décrit un pas d'une marche de concert sur le graphe $G(6, 3, 3)$, ainsi que les sommets vivants lors de cette marche. Par exemple, il y a initialement deux sommets vivants, et il ne reste qu'un sommet vivant après deux pas.
Nous proposons de calculer la marche de concert en maintenant deux structures de données : la première contenant l'état du graphe, c'est-à-dire les agents présents dans chaque sommet ; et la seconde contenant l'ensemble des sommets vivants.

Question 6 *Implémenter l'algorithme réalisant une marche de concert à l'aide de l'approche proposée ci-dessus. Utiliser cette implémentation pour calculer le nombre de sommets vivants après L pas de la marche de concert dans les graphes suivants :*

- ***a)*** $G(6, 3, 3), L = 4$
- ***b)*** $G(1\,234, 10, 3), L = 100$
- ***c)*** $G(10\,001, 22, 10), L = 50\,000$
- ***d)*** $G(100\,001, 40, 10), L = 100\,000$



Légende : Les agents présents dans un sommet sont indiqués au dessus ou à côté de celui-ci. Les sommets vivants sont marqués d'un double cercle. Par exemple, avant le premier pas (donc dans le graphe le plus à gauche), les sommets 0 et 5 sont vivants.

FIGURE 2 – Deux pas d'une marche de concert dans le graphe $G(6, 3, 3)$ pour $\widetilde{u_0}$.

```
(* Pour un graphe à [n] sommets, [state] est un tableau de [n] éléments.
   La [i]-ème case de [state] contient une paire [(k,agents)] où :
   - [agents] est la liste des agents présents dans le sommet [i]
   - [k = List.length agents] *)
type state = (int * int list) array
```

```
(* retourne la paire de :
   - la liste des paires de sommets et des agents dans ces sommet
   - la liste des sommets vivants
   après [max_step] pas de la marche de concert sur le graphe [g]. *)
let concurrent_executions
    (strat : strat) (g : graph) (max_step : int) : (int * int list) list *
    ↪  int list
  =
  let n, _, p = g in
  let st : state = Array.init n (fun i -> (1, [i])) in

  (* Tout le monde avance d'un pas si possible.
     La liste [alive] contient la liste des sommets dans lesquels
     des agents sont présents : cette liste mise à jour est
     retournée par la fonction. *)
  let all_step1 (alive : int list) : int list =
    (* liste des agents ayant fait un pas *)
    let moved : (int * int * int list) list =
      List.concat_map (fun i ->
          let k, agents = st.(i) in
          (* on enlève les agents du sommet [i] *)
          let () = st.(i) <- (0,[]) in
          if k = 0 then [] (* personne à déplacer *)
          else
            let j = strat i in (* prochain sommet *)
            [j, k, agents]     (* déplacement ajouté à [moved] *)
        ) alive
    in

    (* applique les déplacements de [moved] dans l'état, en calculant le
       nouveau [alive] *)
    List.fold_left (fun alive (j,k,agents) ->
        (* agents déjà en [j] *)
        let k', agents' = st.(j) in
        let new_k = k + k' in
        let new_agents =
          (* optimisation : on concatène la plus petite des deux listes. *)
          if k < k' then agents @ agents' else agents' @ agents
        in
        st.(j) <- (new_k, new_agents);
        if new_k > p j then j :: alive else alive
      ) [] moved
  in

  (* Tout le monde avance de [step] pas si possible. *)
  let rec all_stepn (alive : int list) (step : int) : int list =
    if step = 0 then alive else all_stepn (all_step1 alive) (step - 1)
  in
  let init_alive : int list =
    let rec init (i : int) acc =
      if i < 0 then List.rev acc else
        init (i - 1) (if p i < 1 then i :: acc else acc)
    in
    init (n - 1) []
  in
  let alive = all_stepn init_alive max_step in
  let final_conf =
    Array.mapi (fun i (_, agents) -> i, agents) st |>
    Array.to_list
  in
  (* on élimine les doublons *)
  final_conf, List.sort_uniq Stdlib.compare alive
```

```
let nb_alive (g : graph) ~(l : int) : int =
  let strat = pi0 g in
  let conf, alive = concurrent_executions strat g l in
  List.length alive
```

Question 7 *On note $C(\mathcal{G}, L, s)$ l'ensemble des agents dans le sommet s après L pas de la marche de concert dans le graphe $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \textit{poids})$. Calculer la quantité :*

$$\left(\sum_{s\in\mathcal{N}} \sum_{a\in C(\mathcal{G},L,s)} s \cdot a\right) \ mod\ 1000$$

pour les graphes et nombres de pas suivants :

a) $G(6, 3, 3), L = 4$

b) $G(1\,234, 10, 3), L = 100$

c) $G(10\,001, 22, 10), L = 50\,000$

d) $G(100\,001, 40, 10), L = 100\,000$

```
let sum_agents (g : graph) ~(l : int) : int =
  let strat = pi0 g in
  let conf, _ = concurrent_executions strat g l in
  let sum =
    List.fold_left
      (fun sum (s,agents) ->
         sum + s * List.fold_left (+) 0 agents
      )
      0 conf
  in
  sum mod 1_000
```

Dans les deux questions à préparer à l'oral suivantes, on exprimera les coûts en temps et en mémoire en fonction, entre autres, du nombre de groupes d'agents et du nombre de groupes d'agents vivant au i-ème pas (pour i allant de 0 à L).
Remarque : on ne demande pas de trouver une formule caractérisant le nombre de groupes d'agents et de groupes d'agents vivants après i pas.

Question à développer pendant l'oral 4 *Décrire la structure de données que vous avez utilisée pour : i) stocker l'état du graphe ; et ii) stocker l'ensemble des sommets vivants.*
Donner leur espace mémoire, et le coût en temps pour mettre à jour ces deux structures de données lors de l'exécution d'un pas de la marche de concert.

On utilise :
- un tableau A de n cases où la i^e case contient la liste des agents présents dans le sommet i ;
- et la liste V des sommets vivants.

Soit V' la liste qui contiendra les sommets vivants après le pas de la marche de concert. On utilisera aussi une liste M dans laquelle on stockera (temporairement) des ordres de déplacement : M est une liste de paires $(j, [a_1, \dots, a_k])$ où $[a_1, \dots, a_k]$ sont des agents devant se déplacer en j. Pour effectuer un pas de la marche de concert, il suffit d'itérer sur la liste des sommets vivants (donc $|\mathsf{V}|$ fois), et pour chaque sommet vivant :

(1) de faire avancer tous les agents dans $\mathsf{A}[i]$ d'un pas vers j, où $j = \pi_0(\mathcal{G})(i)$: on vide $\mathsf{A}[i]$, et on stocke l'ordre de déplacement $(j, \mathsf{A}[i])$ dans M (en $\mathcal{O}(1)$).

(2) d'ajouter j à la liste V' des sommets vivants après le pas de la marche de concert (en $\mathcal{O}(1)$).

Après cela, on applique tous les ordres de déplacement dans M : où chaque ordre $(j, [a_1, \dots, a_k])$ coûte $\mathcal{O}(k)$ (car on utilise des listes).

L'application des ordres de déplacement peut être optimisé, par exemple :

(i) en stockant dans A la paire de la liste des agents présents et de la longueur de la liste (idem dans M, on garde en mémoire le nombre d'agents), de façon à ce qu'appliquer l'ordre de déplacement $(j, k, [a_1, \dots, a_k])$ dans $\mathsf{A}[j] = (k', [b_1, \dots, b_{k'}])$ coûte la plus petite des deux listes, c'est à dire $\min(k, k')$. Cela donne un très bon coût amorti, voir question suivante.

(ii) En utilisant une structure de donnée plus complexe pour représenter les ensembles d'agents, qui supporte l'union en $\mathcal{O}(1)$. Par exemples, on pourrait utiliser des arbres.

Au total, la complexité en temps d'un pas est en $\mathcal{O}(|\mathsf{V}| \times d)$, où d est le nombre maximal d'agents dans un sommet vivant (avec l'optim. (ii), on est en $\mathcal{O}(|\mathsf{V}|)$).
La complexité en espace est en $\mathcal{O}(n)$.

Question à développer pendant l'oral 5 *Donner la complexité en temps et mémoire de l'évaluation de L pas de la marche de concert.*

Le nombre maximal d d'agents dans un sommet vivant au i^{e} pas peut être de l'ordre de n. Donc, sans optimisations, on est en $\mathcal{O}(\sum_{i \leq L} |\mathsf{V}_i| \times n)$.
L'optimisation (ii) donne une bien meilleure complexité, en $\mathcal{O}(\sum_{i \leq L} |\mathsf{V}_i|)$.
Une analyse fine de la complexité amortie de (i) montre que celle-ci est efficace. En effet, dans ce cas le coût d'application d'un ordre de déplacement vers un sommet vide est en $\mathcal{O}(1)$: seules les fusions d'agents sont coûteuses. Il y a au plus n fusions d'agents lors d'une marche complète, et une fusion coûte au plus $\mathcal{O}(n)$. On a donc une complexité totale en $\mathcal{O}((\sum_{i \leq L} |\mathsf{V}_i|) + n^2)$.

En bornant chaque $|\mathsf{V}_i|$ par n (ce qui est très grossier, puisqu'après un certain temps, $|\mathsf{V}_i|$ peut être bien plus petit que n), on obtient une complexité en : $\mathcal{O}(L \times n^2)$ sans optimisation, $\mathcal{O}(L \times n + n^2)$ avec (i), et $\mathcal{O}(L \times n)$ avec (ii).

3 Marche et stratégie optimale

Dans cette section, nous allons associer à chaque marche une valeur (entière), puis nous chercherons à déterminer la stratégie optimale à horizon fini L, c'est-à-dire la stratégie dont la marche associée (en partant du sommet 0) est de valeur maximale parmi toutes celles de longueur L.

3.1 Valeur d'une marche

On considère une notion de valeur d'une marche paramétrée par un entier $\alpha \in \mathbb{N}$. Étant donné une marche ρ sur un graphe $\mathcal{G}$, la valeur de ρ pour le paramètre α, que l'on note $\mathsf{valeur}_\alpha(\rho)$, est la somme du poids des sommets par lesquels passe ρ, en ne comptant pas plus de α fois le poids de chaque sommet. Plus précisément :

$$\mathsf{valeur}_\alpha(\rho) = \sum_{s \in \mathcal{N}} \mathsf{poids}(s) \cdot \min(\alpha, \mathsf{count}(s, \rho))$$

où $\mathsf{count}(s, \rho)$ compte le nombre de passage de ρ dans un sommet s.

Question 8 *Implémenter la fonction* $\mathsf{valeur}_\alpha(\cdot)$, *et l'utiliser pour calculer*

$$\mathsf{valeur}_\alpha(\rho) \mod 999$$

pour la marche ρ *de longueur* L *partant du sommet* 0 *et suivant la stratégie* $\pi_0(\mathcal{G})$, *pour les graphes* $\mathcal{G}$, *paramètres* α *et longueurs de marche* L *suivants :*

- ***a)*** $G(6, 3, 10), \alpha = 2, L = 6$
- ***b)*** $G(123, 7, 10), \alpha = 20, L = 100$
- ***c)*** $G(1\,234, 10, 10), \alpha = 1000, L = 100\,000$
- ***d)*** $G(10\,001, 22, 10), \alpha = 10\,000, L = 10\,000\,000$
- ***e)*** $G(100\,001, 40, 10), \alpha = 100, L = 1\,000\,000\,000\,000$
- ***f)*** $G(200\,002, 50, 10), \alpha = 100, L = 1\,000\,000\,000\,000$

Question à développer pendant l'oral 6 *Détailler votre algorithme et donner sa complexité en temps et en mémoire.*

Une marche suivant une stratégie sans mémoire suffisamment longue finit toujours dans un cycle. Un cycle élémentaire est de longueur au plus n, et après α répétitions, il est inutile de continuer à exécuter la marche. Donc il suffit de simuler la marche pour au plus $(n+1) \times \alpha$ pas (n pas pour atteindre le cycle, α répétitions du cycle).
De plus, il est inutile de simuler la marche pour plus de L pas.
Complexité pour n sommets et m arcs :

- temps : $\mathcal{O}(\min(L, n \times \alpha))$
- espace : $\mathcal{O}(1)$

3.2 Stratégie avec mémoire optimale

Nous allons maintenant chercher à déterminer la stratégie optimale parmi l'ensemble des marches d'une certaine longueur L.
Pour l'instant, nous n'avons considéré que des stratégies sans mémoire. Dans cette section, nous allons nous intéresser à des stratégies plus complexes, les stratégies avec mémoire. Une stratégie avec mémoire π_{h} pour un graphe $(\mathcal{N}, \mathcal{E}, \mathsf{poids})$ est une fonction qui choisit dans quel sommet se

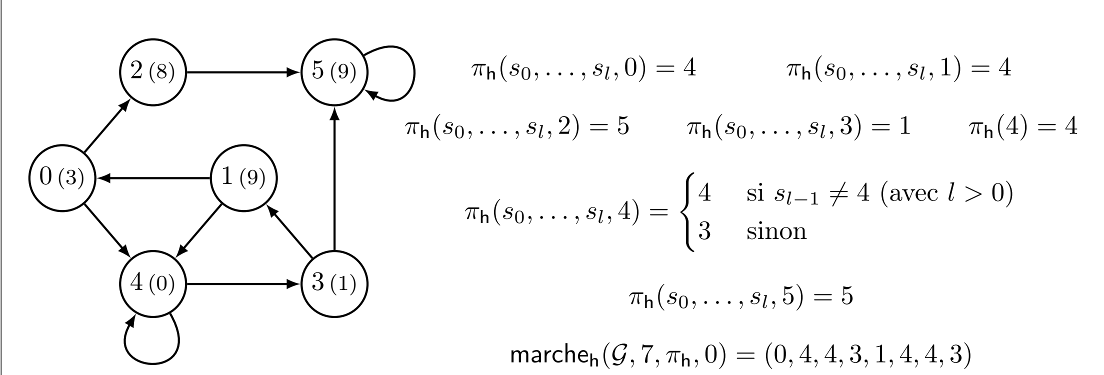
```ocaml
let value_min (a : int) (count : int array) (g : graph) : int =
  let n, _, p = g in
  let sum = ref 0 in
  (* pour chaque sommet [i] de [g], ajoute [min a (count(i))] *)
  Array.iteri (fun i c ->
      sum := !sum + p i * min a c
    ) count;

  !sum

(* version optimisée, qui profite de quand [a] est petit. *)
let run_value_opt ~(a : int) (g : graph) (strat : strat) ~(l : int) : int =
  let n,_,p = g in
  let count = Array.make n 0 in (* compteur du nombre de passages dans chaque
  ↪  état *)

  let s = ref 0 in               (* sommet courant, pas encore comptabilisé *)
  let cycle : int option ref = ref None in       (* début d'un cycle *)
  let i = ref 0 in               (* compteur de la longueur du run *)

  (* on cherche un cycle *)
  while (!i <= l && !cycle = None) do
    if count.(!s) > 0 then cycle := Some !s;     (* on a trouvé un cycle, on
    ↪  sort *)
    count.(!s) <- count.(!s) + 1;
    s := strat !s;
    incr i;
  done;

  let cycle : int = Option.get !cycle in
  let cpt_cycle = ref 1 in

  (* on fait au plus [a] tour dans le cycle *)
  while (!i <= l && !cpt_cycle < a) do
    count.(!s) <- count.(!s) + 1;
    s := strat !s;
    incr i;
    if !s = cycle then incr cpt_cycle; (* on a fait un cycle de plus *)
  done;

  (* sanity check, on vérifie qu'on a bien compté tous les sommets du cycles
  ↪  *)
  for _ = 0 to n do
    s := strat !s;
  done;

  value_min a count g
```

Figure 3 – Code OCaml correction question 8.

déplacer en fonction de l'historique des sommets déjà empruntés. Plus précisément, une stratégie avec mémoire π_{h} est une fonction associant à toute marche non-vide $\rho = s_0, \ldots, s_l$ de $\mathcal{G}$ un successeur de s_l :

$$(s_l, \pi_{\mathsf{h}}(s_0, \ldots, s_l)) \in \mathcal{E} \quad \text{pour toute marche } s_0, \ldots, s_l \text{ de } \mathcal{G}$$

On note $\mathsf{marche}_{\mathsf{h}}(\mathcal{G}, L, \pi_{\mathsf{h}}, s)$ la marche de longueur L partant d'un sommet s et suivant la stratégie avec mémoire π_{h} dans le graphe $\mathcal{G}$. Un exemple de stratégie avec mémoire et de marche est donné Figure 4.



À droite, une stratégie avec mémoire π_{h} restant dans le sommet 4 deux fois de suite, et la marche de longueur 7 suivant π_{h} et partant de 0.

FIGURE 4 – Exemple de stratégie avec mémoire et de marche sur le graphe $G(6, 3, 10)$ pour $\widetilde{u_0}$.

Une stratégie π_{h}^o est optimale pour $\mathsf{valeur}_\alpha(\cdot)$ à horizon fini $L \in \mathbb{N}$ si la marche associée est de valeur maximale parmi toutes les marches de longueur L au départ du sommet 0, c'est-à-dire :

$$\pi_{\mathsf{h}}^o = \underset{\pi_{\mathsf{h}}}{\operatorname{argmax}} \Big(\mathsf{valeur}_\alpha(\mathsf{marche}_{\mathsf{h}}(\mathcal{G}, L, \pi_{\mathsf{h}}, 0)) \Big).$$

Question à développer pendant l'oral 7 *Donner un graphe $\mathcal{G}$ et un horizon $L \in \mathbb{N}$ tels qu'il existe une stratégie avec mémoire sur $\mathcal{G}$ de valeur strictement meilleure que toute stratégie sans mémoire sur $\mathcal{G}$ (à horizon L, au départ de 0).*

Le graphe $0, 1, 2$ de poids respectifs 1, 10, et 0, avec les transitions $0 \to 0$, $0 \to 1$, $1 \to 2$, et $2 \to 2$.
Alors, pour $L = 5$ par exemple, la meilleure stratégie au départ de 0 est d'attendre en 0 aussi longtemps que possible (ici, 4 pas), avant de passer en 1 pour obtenir le gain de 10 (qui ne peut être obtenu qu'une fois).

Question 9 *En énumérant toutes les marches de longueur L, calculer la valeur selon $\mathsf{valeur}_\alpha(\cdot)$ de la stratégie avec mémoire optimale au départ du sommet 0 pour les graphes $\mathcal{G}$, horizons L et*

paramètres α suivants :

a) $G(6, 3, 10), \alpha = 1, L = 6$

b) $G(6, 3, 10), \alpha = 6, L = 6$

c) $G(1\,234, 10, 10), \alpha = 2, L = 9$

d) $G(10\,001, 22, 10), \alpha = 2, L = 4$

Question à développer pendant l'oral 8 *Évaluer la complexité en temps et en mémoire de l'exécution de votre algorithme.*

Complexité pour n sommets, au plus M arcs sortant par sommet, et longueur de marche L.
On énumère toutes les marches de longueur L au départ de 0 dans le graphe. Il y a au plus M^L telles marches.
On n'a besoin de stocker que le chemin courant qu'on est en train d'énumérer, d'au plus L sommets.

- temps : $\mathcal{O}(M^L)$
- espace : $\mathcal{O}(L)$

3.3 Stratégie avec mémoire optimale : cas $\alpha > L$

Dans le cas où $\alpha > L$, la valeur $\mathsf{valeur}_\alpha(\rho)$ d'une marche $\rho = s_0, \ldots, s_L$ est la somme des poids des sommets de ρ, et peut être écrite :

$$\mathsf{valeur}_\alpha(s_0, \ldots, s_L) = \sum_{0 \leq i \leq L} \mathsf{poids}(s_i)$$

Puisque la valeur précise de α n'a plus d'importance quand $\alpha > L$, on pose $\mathsf{valeur}(\rho) = \mathsf{valeur}_{L+1}(\rho)$ où L est la longueur de ρ.

Question 10 *Calculer la valeur selon $\mathsf{valeur}(\cdot)$ de la stratégie avec mémoire optimale <u>au départ du sommet 0</u> pour les graphes $\mathcal{G}$ et horizons L suivants :*

a) $G(6, 3, 10), L = 6$

b) $G(6, 3, 10), L = 200$

c) $G(1\,234, 10, 10), L = 40$

d) $G(10\,001, 22, 10), L = 20$

Question à développer pendant l'oral 9 *Détailler votre algorithme et donner sa complexité en temps et en mémoire.*

Par programmation dynamique. On calcule le tableau $a[l][i]$ qui contient la valeur de meilleure marche de longueur l au départ de i.
$a[l+1][i]$ se calcule en itérant sur tous les j tels que $i \to j$, donc en temps $\mathcal{O}(M)$.
On a seulement besoin de stocker la l^{e} ligne du tableau a pour calculer la $l+1^{\text{e}}$ ligne.
Complexité pour n sommets et au plus M arcs sortants par sommet :

- temps : $\mathcal{O}(L \times n \times M)$
- espace : $\mathcal{O}(n)$ (si on ne stocke pas les chemins)

```ocaml
(* la valeur d'un run [rho] dans [g] avec le coefficient [a]
   [\sum_i p i * min(occ[i],a)] *)
let run_value_min (a : int) (g : graph) (rho : run) : int =
  let n, e, p = g in
  let count = Array.make n 0 in

  (* compte combien de fois la marche [rho] est passée par
     chaque état *)
  List.iter (fun i -> count.(i) <- count.(i) + 1) rho;

  value_min a count g

(* Meilleur run pour [run_value] dans [g] à horizon fini [l] au départ
   de [0].
   Énumère toutes les marches.
   Pré-requis : [forall r, run_value g r > 0] *)
let best_run_enum ~run_value ~(l : int) (g : graph) : int * run =
  let n, e, p = g in

  (* Retourne la meilleure marche entre :
     - la meilleure marche de longueur [l] étendant [List.rev part_run]
     - la marche dans [acc]

     Invariants :
     - [i = List.hd part_run] (donc [part_run] est non-vide)
     - si [acc = (v,r)], alors [v = run_value g r]

     Tail-recursive *)
  let rec best_run
      (part_run : run) (i : int) (l : int) (acc : (int * run))
    : (int * run)
    =
    if l = 0 then begin
      let run = List.rev part_run in
      let value = run_value g run in
      let value', run' = acc in
      if value > value' then (value, run) else (value',run')
    end
    else
      List.fold_left (fun acc j ->
          best_run (j :: part_run) j (l - 1) acc
        ) acc (e.(i))
  in

  if l = 0 then 0, [] else
    (* meilleur run de longueur [l] étendant [0]. *)
    best_run [0] 0 l (-1, [])

let best_run_min ~a = best_run_enum ~run_value:(run_value_min a)
```

Figure 5 – Code OCaml correction question 9.

```ocaml
(* la valeur d'un run [rho] dans [g]
   [\sum_j p (rho j)] *)
let run_value_sum (g : graph) (rho : run) : int =
  let _, _, p = g in
  List.fold_left (fun sum j -> sum + p j) 0 rho

(* Meilleur run pour [run_value_sum] dans [g] à horizon fini [l]
   au départ de [0].
   Programmation dynamique. *)
let best_run_sum (g : graph) ~(l : int) : int * run =
  let n, e, p = g in

  (* meilleures marches de longueur [1] *)
  let a1 = Array.init n (fun i -> p i, [i]) in
  let a2 = Array.make n (-1, []) in

  (* [src] et [dst] sont des tableaux de longueur [n] :
     - [src.(i)] contient la meilleure marche (et sa valeur),
       de longueur [l] au départ de [i].
     - [dst.(i)] doit être remplit de même, mais pour les marches
       de longueur [l + 1]. *)
  let doit1 ~(src : (int * run) array) ~(dst : (int * run) array) =
    (* pour chaque sommet [i] *)
    Array.iteri (fun i _ ->
        (* itère sur les sommets [j] tel que [i → j], en cherchant la
           meilleur marche *)
        let value, run =
          List.fold_left (fun (value,run) j ->
              let valuej,runj = src.(j) in
              if valuej > value then (valuej,runj) else (value,run)
            ) (-1,[]) e.(i)
        in
        dst.(i) <- p i + value, i :: run
      ) dst
  in

  (* Similaire à [doit1], mais avance de [l] pas :
     - [src] doit contenir les meilleures marche de longueur [l]
     - retourne le tableau avec les meilleures marche de longueur [l + h]. *)
  let rec doit ~src ~dst (l : int) : (int * run) array =
    if l = 0 then src else
      let () = doit1 ~src ~dst in
      doit ~src:dst ~dst:src (l - 1) (* inverse les tableaux *)
  in

  if l = 0 then 0, [] else begin
    let res = doit ~src:a1 ~dst:a2 l in
    res.(0)
  end
```

FIGURE 6 – Code OCaml correction question 10.

3.4 Stratégie optimale pour des marches infinies

Nous considérons maintenant des marches infinies sur un graphe $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathsf{poids})$. Une marche infinie est une liste de sommets $(s_i)_{i\in\mathbb{N}}$ reliés par des arcs, c'est-à-dire telle que $(s_i, s_{i+1}) \in \mathcal{E}$ pour tout $i \in \mathbb{N}$. Plus précisément, nous allons nous intéresser aux marches ultimement cycliques. Une marche infinie est ultimement cyclique si elle est de la forme :

$$\underbrace{s_0, \ldots, s_L}_{\text{préfixe fini}}, \underbrace{t_0, \ldots, t_K}_{\text{cycle}}, \underbrace{t_0, \ldots, t_K}_{\text{cycle}}, \ldots$$

c'est-à-dire si la marche commence par un *préfixe fini* $s_0, \ldots, s_L$ et se termine par un *cycle* $t_0, \ldots, t_K$ se répétant un nombre infini de fois.
La valeur d'une telle marche infinie $\mathsf{valeur}((s_i)_{i\in\mathbb{N}})$ ne peut pas être la somme des poids des sommets s_i, puisque cette somme risque d'être divergente. À la place, nous considérerons que la valeur d'une marche infinie ultimement cyclique est la valeur moyenne des poids des sommets du cycle final, c'est-à-dire que si la marche $(s_i)_{i\in\mathbb{N}}$ finit par répéter le cycle $t_0, \ldots, t_K$, alors

$$\mathsf{valeur}_\infty((s_i)_{i\in\mathbb{N}}) = \frac{\sum_{0\leq i\leq K} \mathsf{poids}(t_i)}{K+1}$$

Noter que cette valeur est bien défini, car elle ne dépend pas du cycle choisi.

On pose $H(n, M, p)$ le graphe obtenu à partir de $G(n, M, p)$ en remplaçant tout arc d'un sommet i vers lui-même par un arc de i vers $(i + 1 \mod n)$. Ces graphes sont sans boucles. Un exemple de tel graphe est donné Figure 7.

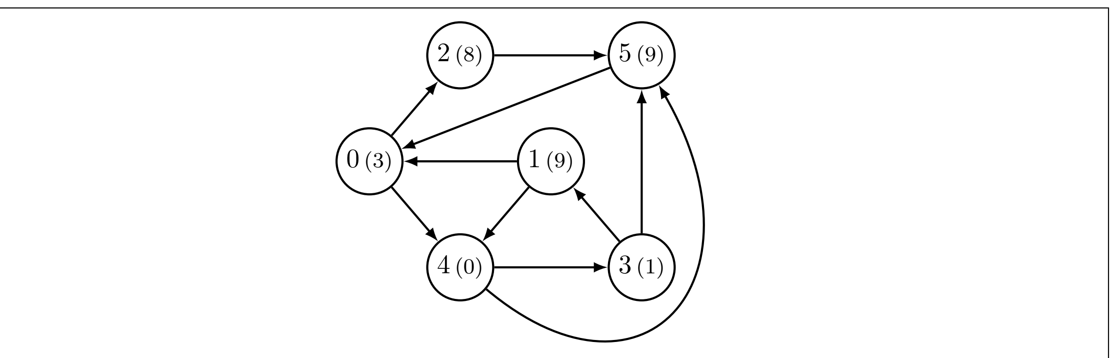


FIGURE 7 – Le graphe sans boucles $H(6, 3, 10)$ pour $\widetilde{u_0}$.

Dans la question suivante, on donnera la valeur d'une marche infinie avec 2 chiffres après la virgule.

Question 11 *Calculer la valeur selon* $\mathsf{valeur}_\infty(\cdot)$ *de la marche infinie ultimement cyclique (au départ de n'importe quel sommet) de valeur maximale dans les graphes $\mathcal{G}$ (sans boucles) suivants :*

a) $H(6, 3, 10)$ ***b)*** $H(10, 3, 10)$ ***c)*** $H(40, 4, 10)$ ***d)*** $H(99, 7, 10)$

Question à développer pendant l'oral 10 *Détailler votre algorithme et donner sa complexité en temps et en mémoire.*

```ocaml
let best_omega_run_any (g : graph) : float =
  let n, e, p = g in
  (* [a.(i).(j)]: value of the best path of length exactly [l] from
     [i] to [j], excluding [j]'s weight *)
  let a  = Array.init n (fun j -> Array.make n ([],min_int)) in
  (* copy, to store results at step [l+1] *)
  let a' = Array.init n (fun j -> Array.make n ([],min_int)) in
  (* initialise [a] for [l=1] *)
  for i = 0 to n-1 do
    List.iter (fun j -> a.(i).(j) <- ([i; j],p i)) e.(i)
  done;
  let best_cycle = ref (-1., []) in

  for l = 2 to n do
    (* compute in [a'] the values of the best path of length [l+1] *)
    for i = 0 to n-1 do
      for j = 0 to n-1 do
        let bestp, best =
          List.fold_left (fun (bestp, best) i0 ->
              let i0_j, val_i0_j = a.(i0).(j) in
              let val_i_i0_j = p i + val_i0_j in
              if best < val_i_i0_j && val_i0_j <> min_int then
                (i :: i0_j, val_i_i0_j)
              else
                (bestp, best)
            ) ([],min_int) e.(i)
        in
        a'.(i).(j) <- (bestp, best)
      done;
    done;
    (* copy [a'] into [a] *)
    for i=0 to n-1 do for j=0 to n-1 do a.(i).(j) <- a'.(i).(j) done; done;
    for i = 0 to n-1 do
      let i_i, val_i_i = a.(i).(i) in
      let val_cycle = float_of_int val_i_i /. float_of_int l in
      if val_cycle > fst !best_cycle && val_i_i <> min_int then
        best_cycle := val_cycle, i_i
    done;
  done;

  let val_cycle, _ = !best_cycle in
  val_cycle

let move_self_loops (g : graph) : graph =
  let n, e, p = g in
  let new_e =
    Array.mapi (fun i l -> List.filter (fun j -> j <> i) l) e
  in

  Array.iteri (fun i l ->
      if List.mem i l then
        let next_i = (i+1) mod n in new_e.(i) <- next_i :: new_e.(i)
    ) e;

  (n, new_e, p)
```

```ocaml
let graph_nl ~n ~m ~p = move_self_loops (graph ~n ~m ~p)
```

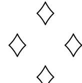
Figure 8 – Code OCaml correction question 11.

Par programmation dynamique aussi. Le tableau $A[l][i][j]$ contient la valeur de la meilleure marche de longueur l de i à j, en excluant le poids de j. On utilise une valeur spéciale $\bot$ s'il n'y a pas de telle marche.
Alors $A[l+1][i][j]$ se calcule à partir des $A[l][i_0][j]$ pour $i \to i_0$, donc en $\mathcal{O}(M)$.
Il est suffisant de regarder les cycles élémentaires, donc de calculer $A[l]$ pour l allant jusqu'à n.
On a donc trois boucles imbriquées, chacune itérée n fois.
À la fin, on itère sur tout les $A[l][i][i]$ pour l de 1 à n, en cherchant le meilleur cycle.

Le tableau A est de taille $\mathcal{O}(n^3)$. Comme il suffit de se rappeler de $A[l]$ pour calculer $A[l+1]$, et en calculant le meilleur cycle au fur et à mesure, on peut descendre à $\mathcal{O}(n^2)$.

Complexité pour n sommets et au plus M arcs sortants par sommet :

— temps : $\mathcal{O}(n^3 \times M)$
— espace : $\mathcal{O}(n^2)$ (si on ne stocke pas les chemins)



Fiche réponse type : Des marches dans des graphes

$\widetilde{u_0}$: 104

Question 1

a) 464

b) 281

c) 919

d) 390

Question 2

a) 5

b) 12

c) 20

d) 20

Question 3

a) 10

b) 316

c) 3583

d) 36938

Question 4

a) 582

b) 629

c) 820

d) 92

Question 5

a) 25

b) 284

c) 382

d) 704

Question 6

a) 1

b) 63

c) 8

d) 57

Question 7

a) 61

b) 729

c) 651

d) 453

Question 8

a) 29

b) 108

c) 310

d) 41

e) 313

f) 461

Question 9

a) 30

b) 56

c) 78

d) 37

Question 10

a) 56

b) 1802

c) 363

d) 169

Question 11

a) 6.67

b) 4.75

c) 7.00

d) 7.57