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

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

Coefficients (en pourcentage du total des points du concours Informatique) :
- ENS de Lyon : 16,9 %
- ENS Paris-Saclay : 13,2 %
- ENS Ulm : 13,3 %

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

Contenu de ce document

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

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

Organisation de l'épreuve

L'objectif de cette épreuve est d'évaluer la capacité à mettre en œuvre une chaîne complète de résolution d'un problème informatique, à savoir la construction d'algorithmes, le choix de structures de données, leur implémentation, et l'élaboration d'arguments mathématiques pour justifier ces décisions. Le déroulement de l'épreuve est le suivant : un travail sur machine d'une durée de 3 h 30, immédiatement suivi d'une présentation orale pendant 24 minutes.

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

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

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

Les questions écrites demandent de calculer certaines valeurs, souvent numériques, bien que d'autres types tels que des chaînes de caractères soient également possibles. Chaque question requiert l'implémentation d'un algorithme et son utilisation sur les entrées générées au début du sujet, pour calculer les valeurs demandées. Une question est

généralement divisée en sous-questions pour des entrées de plus en plus grandes, ce qui permet de tester l'efficacité de l'algorithme mis en œuvre. Les réponses sont à inscrire sur une fiche-réponse, qui sera remise au jury à l'issue de la partie pratique de l'épreuve.

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

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

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

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

Conseils et remarques générales

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

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

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

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

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

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

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

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

Les bornes de complexité d'algorithmes par force brute ont semblé peu claires à certaines personnes : nous avons parfois vu des réponses fantaisistes donnant par exemple une majoration du nombre de chemins dans un graphe linéaire en son nombre de sommets, et

des candidats ou candidates s'étonner qu'un algorithme exponentiel ne permette pas de conclure sur de grandes valeurs.

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

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

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

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

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

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

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

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

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

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

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

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

Sujet 1 : Affectation de maisons

Ce sujet portait sur les algorithmes d'affectation de ressources, dont certains sont utilisés pour affecter les étudiants dans l'enseignement supérieur. On explorait ainsi trois scénarios où l'objectif est d'affecter n maisons à n personnes : avec des préférences (chaque personne a un ordre de préférence sur les maisons), avec des préférences et une répartition initiale (chaque personne possède initialement une maison), et avec des préférences et des priorités (chaque maison a un ordre de priorité sur les personnes).

Le sujet se décomposait en 7 parties : la 1re partie préliminaire détaillait la génération des données, les parties 2 et 3 (indépendantes) présentaient deux algorithmes tenant compte des préférences des personnes, les parties 4 et 5 (indépendantes) proposaient deux algorithmes tenant également compte des priorités des maisons, et enfin les parties 6 et 7 (indépendantes) étudiaient les propriétés des affectations calculées.

Partie 1. La première partie commençait, comme souvent, par la génération des données nécessaires pour les parties suivantes. Les questions Q1 et Q2 permettaient de générer des permutations aléatoires avec une variante de l'algorithme de Fisher-Yates. Il n'était pas nécessaire de comprendre la distribution obtenue pour résoudre le reste du sujet (permutation identité avec $d=0$, et permutation uniforme avec $d=n$). La question Q3 introduisait la mesure du rang total d'une affectation, et permettait de vérifier la bonne compréhension des notations du sujet. Ces trois questions d'échauffement ont été bien réussies.

Partie 2. La deuxième partie présentait l'algorithme des dictateurs successifs, où les personnes arrivent tour à tour et choisissent leur maison préférée parmi les maisons restantes. L'implémentation (question Q4) ne présentait pas de difficulté particulière (un pseudo-code étant fourni dans le sujet). La question QO1 demandait la complexité en temps de l'algorithme, dans le pire des cas, en fonction de n. Le sujet mentionnait la complexité dans le pire des cas pour inviter les candidats à ne surtout pas réfléchir à la complexité en moyenne (les données étant pseudo aléatoires). Certains candidats ont perdu

un temps précieux en présentant une instance sur laquelle la complexité de l'algorithme est maximale, alors qu'une simple borne asymptotique de $O(n^2)$ opérations pour deux boucles imbriquées était attendue.

Partie 3. La troisième partie présentait l'algorithme des cycles d'échange, permettant de réaffecter les maisons, à partir d'une affectation initiale, en tenant compte des préférences de chaque personne. Dans la question QO2, la moitié de points a été donnée pour la justification de l'existence d'un cycle, et l'autre moitié pour la procédure permettant de le calculer. Certains candidats ont perdu du temps en essayant de présenter des preuves trop formelles (injectivité d'une application, ou récurrence forte), alors qu'un argument plus simple était suffisant (un chemin infini dans un graphe fini boucle forcément). Pour calculer le cycle, la solution attendue était une boucle en maintenant un tableau des sommets déjà parcourus. Un mauvais choix de structure de donnée (liste des sommets déjà parcourus) était pénalisé s'il menait à une complexité quadratique. Les candidats ayant présenté une solution plus générale (parcours en profondeur) n'ont pas été pénalisés. L'implémentation (question Q5) de l'algorithme des cycles d'échange à été la première difficulté du sujet, et environ deux tiers des candidats sont parvenus à une implémentation. Nous rappelons l'importance de décomposer le code en fonctions (comme la recherche du cycle), que l'on peut tester individuellement. Une implémentation naïve de l'algorithme en $O(n^3)$ permettait de calculer toutes les réponses numériques de la question Q5, et une analyse correcte de sa complexité donnait deux tiers des points à la question QO3. Un algorithme quadratique et une analyse correcte de sa complexité était nécessaire pour obtenir tous les points à la question QO3.

Partie 4. La quatrième partie abordait un nouveau scénario, et proposait une adaptation de l'algorithme des cycles d'échange lorsque chaque maison a un ordre de priorité sur les personnes. La question QO4 amenait à réfléchir à l'algorithme sur deux types d'instances (priorités identiques, ou premiers choix distincts). Exécuter l'algorithme à la main sur de petites instances ($n \leq 5$) permettait d'observer assez rapidement des similarités avec les algorithmes étudiés dans les parties précédentes. Certains candidats n'ont pas eu ce réflexe, préférant utiliser leur implémentation de l'algorithme, et n'ont pas remarqué les propriétés des instances proposées. La question Q6 demandait l'implémentation de l'algorithme des cycles d'échange généralisé, dont le pseudo-code n'était pas fourni. Cependant, une bonne modularité du code permettait de réutiliser certaines fonctions déjà écrites (comme la recherche du cycle).

Partie 5. La cinquième partie présentait l'algorithme d'acceptation différée, utilisé dans l'enseignement supérieur pour affecter les étudiants aux écoles et universités. La question QO5 demandait une borne supérieure sur le nombre d'itérations de la boucle principale, et a été réussie par la majorité des candidats, parfois après plusieurs indices (pourquoi la boucle ne peut-elle pas être infinie ? peut-on trouver un variant de boucle ?). Environ la moitié des candidats sont parvenus à implémenter l'algorithme d'acceptation différée, avec un large éventail de complexités ($O(n^4)$, $O(n^3)$, ou $O(n^2)$, dans le pire des cas), qui permettaient toutes d'obtenir les réponses numériques à la question Q7 (les données pseudo aléatoires n'étant pas le pire des cas). Pour la question QO6, une analyse de complexité correcte d'un algorithme de complexité en $O(n^4)$ donnait la moitié des points (deux tiers des candidats), une complexité en $O(n^3)$ donnait environ trois quarts des points (un quart des candidats), et une complexité en $O(n^2)$ était nécessaire pour obtenir tous les points (aucun candidat).

Partie 6. La sixième partie présentait la propriété de Pareto-optimalité, que l'on admettait être obtenue avec les algorithmes présentés dans les parties 2 et 3 (et également 4). Dans la question Q8, plusieurs optimisations étaient nécessaires pour calculer toutes les réponses numériques. La plupart des candidats ayant abordé les questions Q8 et QO7 ont utilisé l'algorithme des dictateurs successifs sur tous les ordres possibles, en éliminant plus

ou moins efficacement les doublons, ce qui permettait d'obtenir les réponses numériques pour les premières valeurs de n.

Partie 7. Enfin, la septième partie présentait la propriété de stabilité, que l'on admettait être obtenue avec l'algorithme présenté dans la partie 5. Dans la question Q9, de nombreuses optimisations étaient nécessaires pour calculer toutes les réponses numériques. Les quelques candidats ayant traité la question Q9 ont généré toutes les affectations et vérifié à postériori lesquelles sont stables. Dans la question QO8, un seul candidat à proposé de générer les affectations avec une fonction récursive qui d'arête d'explorer une branche une fois qu'une instabilité est détectée.

Écrit	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Tous les points	100	97	94	94	65	42	48	0	0
Réponses partielles	100	100	97	100	71	48	58	39	10
Réussite moyenne	100	99	96	96	68	44	51	13	3

Oral	QO1	QO2	QO3	QO4	QO5	QO6	QO7	QO8
Tous les points	97	74	3	12	42	0	3	3
Réponses partielles	100	100	100	97	97	93	61	13
Réussite moyenne	99	93	67	58	72	54	36	8

TABLE 1. Pourcentages tronqués de réponses correctes, partielles et moyennes des points à chaque question du sujet 1.

SUJET 2 : CALCULS ET TABLEURS

Le sujet portait sur la notion de calcul incrémental dans le cadre de tableurs. On explorait ainsi la notion de graphe de dépendances et de tri topologique afin de trouver un ordre d'évaluation, mais aussi pour développer des méthodes pour ne recalculer que le strict nécessaire, après des petits changements dans les données initiales.

La partie écrite commence, comme habituellement, par la génération des données. Les trois premières questions y sont consacrées, permettant de vérifier si la génération a bien été implantée. Ces questions simples ont été majoritairement réussies, sans atteindre un taux de réussite de 100%. Une erreur à ces questions permettait rarement d'obtenir des bons résultats dans la suite du sujet. Bien qu'une erreur soit possible dans la fiche réponse-type, il est bien plus probable que l'erreur se trouve du côté de la candidate ou du candidat qui doit alors s'assurer d'avoir lu l'énoncé avec attention. La question 4 demandait de construire le graphe de dépendances tel qu'il était décrit dans le sujet. Malgré les exemples, il a souvent été construit à l'envers ce qui permettait malgré tout de traiter le sujet mais pas toujours de la façon la plus naturelle. Nous n'avons pas pénalisé cette approche, si ce n'est par le fait que certains résultats se sont retrouvés faux. La question 5 demandait d'établir un tri topologique puis de s'en servir pour évaluer le tableur. Le sujet décrivait succinctement un algorithme pour effectuer le tri, et détecter les éventuels cycles au passage. Trop souvent, les candidates et candidats ont préféré implanter leur propre algorithme sans suivre les indications du sujet, et si dans certains cas cela donnait des réponses correctes, d'autres ont tout simplement oublié la détection de cycle, typiquement en se limitant à deux types de marqueurs, par rapport aux trois suggérés par le sujet. Plusieurs ont également ignoré le tri pour évaluer le tableur. On remarquera par ailleurs que cette question semble avoir été l'obstacle principal avec un taux de réussite soudainement très faible. Peu ont traité les questions suivantes. Les questions 6 et 7 traitaient du calcul incrémental à proprement parler. Nous proposions une approche divisée en deux phases : une première dite d'invalidation (Q6) qui consiste à marquer les

cellules qui doivent être recalculées après un changement, et une de réparation (Q7) qui consiste à appliquer les changements, en ne recalculant que les cellules invalidées. La question 6 a été globalement réussie par celles et ceux qui l'ont entreprise, à ceci près qu'un petit piège se trouvait dans le sujet. En effet, il pouvait arriver dans le tableur qu'une cellule fasse référence deux fois à la même autre cellule, auquel cas le graphe de dépendance ne devait alors présenter qu'un seul arc et non pas deux. Nous nous sommes assurés que cette situation se présentait pour l'ensemble des u0 afin de ne pénaliser personne. La question 7 a été traitée par un seul candidat et ne présentait pas de difficulté particulière une fois la précédente correctement implémentée. La question 8 se concentrait sur le cas de grande feuilles de calculs, mais pour lesquelles on s'intéressait à un faible nombre de valeurs. Il fallait donc penser à ne pas tout stocker inutilement (y compris le tableur) pour éviter une explosion en mémoire. Un grand nombre de valeurs de la suite u était nécessaire mais une remarque au début du sujet indiquait que la suite était cyclique et donc qu'il n'était pas nécessaire de stocker des millions de valeurs. Malheureusement, cette remarque semble être passé inaperçue auprès de beaucoup. Un seul candidat a su traiter cette question dans son entièreté. Pour finir, la question 9 demandait de calculer le nombre de cellules qui avaient plus de n successeurs (dans le sens du graphe de dépendances). Afin d'obtenir des résultats pour toutes les valeurs demandées, plusieurs optimisations étaient nécessaires. Elle n'a semble-t-il pas été traitée.

Pour la partie orale, la première question de complexité a bien été réussie dans l'ensemble, mais les candidates et candidats qui ont su y répondre efficacement ont eu plus de temps pour les autres questions plus intéressantes. Nous sommes déçu des réponses à la question 2 sur le choix de la structure de données pour représenter le graphe. Si toutes et tous ont fait le choix le plus judicieux, peu ont su totalement le justifier, beaucoup arguant un choix plus "simple" au lieu d'avancer la complexité plus faible pour le parcours en profondeur. La complexité du parcours en profondeur (question 3) était souvent connue (mais pas toujours avec le même degré de certitude), mais il était parfois difficile de passer de la taille en fonction du graphe, à la taille en fonction des dimensions du tableur. Nous rappelons que le nombre d'arcs dans un graphe ne saurait être exponentiel en le nombre de nœuds. En effet, vu que nous ne considérions pas d'arcs multiples, il y a au plus deux arcs entre deux nœuds, ce qui donne un nombre au maximum quadratique. Il fallait néanmoins pousser l'analyse plus loin et remarquer que chaque nœud avait au plus deux arcs entrants ce qui donnait un nombre linéaire d'arcs. La complexité de l'évaluation (question 4) était plus simple et mieux maîtrisée. Les questions 5 et 6 s'attardaient sur la complexité des phases d'invalidations et de réparation. Si la complexité en fonction des dimensions du tableur étaient généralement comprise, celle en fonction des valeurs renvoyées l'étaient moins et nécessitaient en général des indications de la part de l'examinateur. La question 7 demandait de remarquer que la génération des données avait un impact fort sur l'efficacité de l'algorithme. Nous attendions des remarques informelles et non plus une preuve. Les personnes qui sont allé le plus loin dans cette question ont remarqué la répartition des cellules, et le nombre moyen d'arcs sortants dans le graphe. On leur a demandé de comparer ce nombre (1) avec une version hypothétique où il vaudrait 2. Pour la question 8, nous attendions des candidates et candidats qu'ils remarquent qu'il était déraisonnable de tout stocker en mémoire, y compris le tableur lui-même ! En effet, il est possible de le recalculer à la volée sans perdre en complexité. Il fallait toutefois penser à la mémoïsation pour éviter de recalculer les valeurs plusieurs fois. Enfin, seuls trois candidats ont traité la question 9 à l'oral et un seul d'entre eux a su exposer l'ensemble des optimisations attendues sans indication.

Écrit	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Tous les points	100	90	83	73	13	13	3	3	0
Réponses partielles	100	100	97	90	37	40	3	10	0
Réussite moyenne	100	97	92	81	23	27	3	5	0

Oral	QO1	QO2	QO3	QO4	QO5	QO6	QO7	QO8	QO9
Tous les points	97	13	30	73	13	13	10	17	3
Réponses partielles	100	100	97	97	90	30	33	37	10
Réussite moyenne	99	65	74	91	59	22	23	30	6

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

Affectation de maisons

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

Durée de l'épreuve : 3 heures 30 minutes

Juin 2024

ATTENTION !

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

Important.

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

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

1 Préliminaires

Nous allons étudier le problème d'affectation de maisons. Étant donné n personnes (numérotées avec $i \in [\![0, n-1]\!]$) et n maisons (numérotées avec $j \in [\![0, n-1]\!]$), l'objectif est d'affecter exactement une maison à chaque personne, en tenant compte des préférences de chaque personne.

La partie 1 est nécessaire pour pouvoir traiter le reste du sujet. Les parties 2 et 3 sont indépendantes, et proposent deux algorithmes qui tiennent compte des ordres de préférences de chaque personne. Les parties 4 et 5 sont indépendantes, et présentent deux algorithmes tenant également compte des ordres de priorité de chaque maison. Enfin, les parties 6 et 7 sont indépendantes et étudient les propriétés des affectations calculées.

Dans la plupart des questions d'implémentation, calculer la réponse pour les dernières valeurs de n nécessite une implémentation efficace. Nous vous suggérons de réfléchir à la complexité en temps avant de commencer l'implémentation. Si votre algorithme n'est pas assez efficace pour calculer certaines réponses, nous vous conseillons d'aborder les questions suivantes, avant de revenir optimiser votre implémentation.

1.1 Notations

Dans le sujet, n est un entier fixé strictement positif. Chaque question d'implémentation vous fournira plusieurs valeurs de n sur lesquelles tester vos algorithmes.

Dans ce sujet, vous serez amenés à manipuler des tableaux. Un **tableau** t de taille n associe à chaque indice $\ell \in [\![0, n-1]\!]$ une valeur $t[\ell]$. On écrira $i \in t$ s'il existe ℓ tel que $t[\ell] = i$, et $i \notin t$ sinon. Enfin, nous utiliserons la notation $[v_0, v_1, \ldots, v_{n-1}]$ pour désigner le tableau t tel que $t[i] = v_i$ pour tout $i \in [\![0, n-1]\!]$.

Une **permutation** de taille n est un tableau qui contient tous les entiers entre 0 et $n-1$ exactement une fois. Le **rang** d'une option $i \in [\![0, n-1]\!]$ dans la permutation est l'indice auquel i apparaît.

1.2 Génération de nombres pseudo-aléatoires

Étant donné u_0, on considère la suite u générée par la relation de récurrence suivante :

$$u_{k+1} = (1\,103\,515\,245 \times u_k + 12\,345) \mod 2^{31}$$

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.

Lors du calcul de la suite u_k, nous vous suggérons de faire attention au problème de dépassement d'entier. On aura souvent besoin de nombreuses valeurs consécutives de la suite u_k. Il est donc conseillé que votre implémentation calcule le tableau de tous les u_k jusqu'à un certain rang choisi soigneusement.

Question 1 *Calculer les valeurs* $u_k \mod 100\,000$, *avec :*

a) $k = 1$ ***b)*** $k = 73$ ***c)*** $k = 1337$ ***d)*** $k = 5\,318\,008$

```python
def compute_u(u0, maxi):
  u = [u0]
  for _ in range(maxi):
    u.append((u[-1] * 1103515245 + 12345) % 2**31)
  return u
u = compute_u(u0, 6000000)

print("1.a", u[1] % 100000)
print("1.b", u[73] % 100000)
print("1.c", u[1337] % 100000)
print("1.d", u[5318008] % 100000)
```

1.3 Génération de permutations pseudo-aléatoires

Dans ce sujet, il vous sera demandé de générer des permutations pseudo-aléatoires. On notera perm(n, d, k) la permutation pseudo-aléatoire de taille n, générée à l'aide des valeurs $u_k, \dots, u_{k+n-2}$ de la manière suivante :

fonction perm(n, d, k)
  initialiser un tableau $t \leftarrow [0, 1, 2, \dots, n-1]$ de taille n.
  pour ℓ *de* 0 *à* $n-2$ **faire**
    définir $r \leftarrow \ell + (u_{k+\ell} \mod \min(d, n-\ell))$
    échanger les valeurs de $t[\ell]$ et $t[r]$
  fin
  renvoyer t
fin

Question 2 *Calculer la permutation* perm$(n, 5, 73)$ *avec :*

a) $n = 3$ ***b)*** $n = 5$ ***c)*** $n = 8$ ***d)*** $n = 10$

```python
def perm(u, n, d, k):
  t = list(range(n))
  for l in range(n-1):
    r = l + u[k+l] % min(d, n-l)
    t[l], t[r] = t[r], t[l]
  return t

def reponse2(n):
  return perm(u, n, 5, 73)

print("2.a", reponse2(3))
print("2.b", reponse2(5))
print("2.c", reponse2(8))
print("2.d", reponse2(10))
```

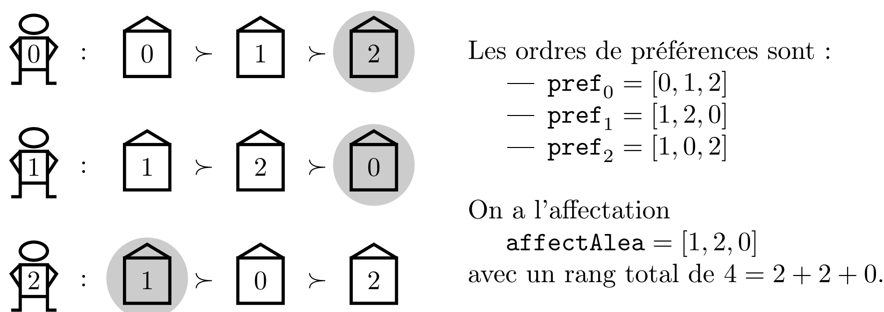
1.4 Ordres de préférences et affectation de maisons

Dans le problème d'affectation de maison, chaque personne a un **ordre de préférence**, qui est une permutation t de taille n, où $t[0]$ est le premier choix, $t[1]$ est le second choix, ..., et $t[n-1]$ est le dernier choix. On note $\texttt{pref}_i$ l'ordre de préférence de la personne i. Dans l'intégralité du sujet, les ordres de préférences seront :

$$\forall i \in [\![0, n-1]\!], \qquad \texttt{pref}_i = \texttt{perm}(n, 50, n \cdot i).$$

Une **affectation** des maisons est une permutation s de taille n, où $s[j]$ est le numéro de la personne qui possède la maison j. Afin de finir notre échauffement, nous allons étudier une **affectation aléatoire** des maisons aux personnes :

$$\texttt{affectAlea} = \texttt{perm}(n, n, n^2)$$



Les ordres de préférences sont :
- $\texttt{pref}_0 = [0, 1, 2]$
- $\texttt{pref}_1 = [1, 2, 0]$
- $\texttt{pref}_2 = [1, 0, 2]$

On a l'affectation
$\texttt{affectAlea} = [1, 2, 0]$
avec un rang total de $4 = 2 + 2 + 0$.

FIGURE 1 – Affectation aléatoire, avec $n = 3$ et $\widetilde{u_0} = 489$.

Enfin, pour mesurer l'efficacité d'une affectation, on regarde pour chaque personne le rang de sa maison dans son ordre de préférence (entre 0 et $n-1$). Le **rang total** d'une affectation est la somme des rangs des maisons dans les ordres de leurs propriétaires. Ainsi, une affectation qui associerait à chaque personne sa maison préférée est de rang total 0. La figure 1 illustre sur un exemple le rang total de l'affectation `affectAlea`.

Question 3 *Calculer le rang total de* `affectAlea` *avec :*

a) $n = 3$ ***b)*** $n = 10$ ***c)*** $n = 100$ ***d)*** $n = 1000$

```
def rang(pref, affect):
  return sum(pref[affect[j]].index(j) for j in range(len(affect)))

def reponse3(n):
  pref = [ perm(u, n, 50, i*n) for i in range(n) ]
  affectAlea = perm(u, n, n, n*n)
  return rang(pref, affectAlea)

print("3.a", reponse3(3))
print("3.b", reponse3(10))
print("3.c", reponse3(100))
print("3.d", reponse3(1000))
```

2 Algorithme des dictateurs successifs

L'affectation aléatoire des maisons ne tient pas compte des préférences de chaque personne. Dans cette partie, nous allons étudier l'**algorithme des dictateurs successifs** : chaque personne choisit successivement sa maison préférée parmi celles qui sont toujours disponibles. Étant donnée une permutation r, on note `affectDictSucc`(r) l'affectation s obtenue par l'algorithme des dictateurs successifs, dans l'ordre r.

fonction `affectDictSucc`(n, r)
  initialiser un tableau $s \leftarrow [\bot, \ldots, \bot]$ de taille n.
  pour *chaque personne* $i = r[0], r[1], ..., r[n-1]$ **faire**
    soit j la maison préférée de i, telle que $s[j] = \bot$.
    affecter $s[j] \leftarrow i$.
  fin
  renvoyer s
fin

Pour que l'algorithme ne favorise pas les personnes en fonction de leur indice, nous allons utiliser un **ordre aléatoire** :

$$\texttt{ordreAlea} = \texttt{perm}(n, n, n^2)$$

La figure 2 illustre sur un exemple l'algorithme des dictateurs successifs, et donne le rang total de l'affectation calculée.

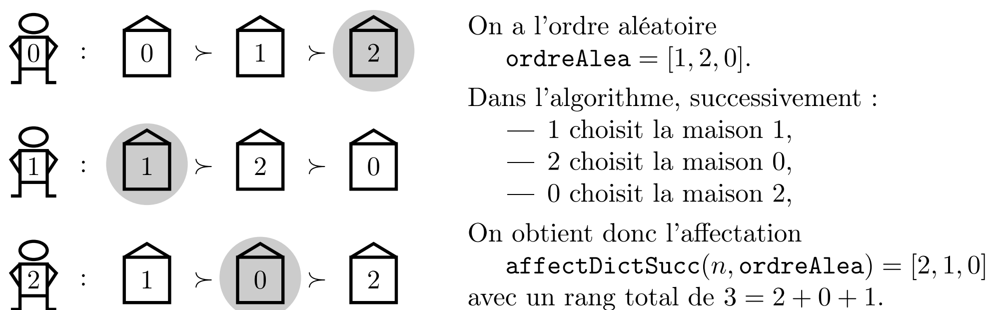


FIGURE 2 – Algorithme des dictateurs successifs, avec $n = 3$ et $\widetilde{u_0} = 489$.

Question 4 *Calculer le rang total de* `affectDictSucc`$(n, \texttt{ordreAlea})$ *avec :*

a) $n = 3$ ***b)*** $n = 10$ ***c)*** $n = 100$ ***d)*** $n = 1000$

```python
def affectDictSucc(pref, ordre):
  n = len(pref)
  affect = [None] * n
  for iPersonne in ordre:
    for jMaison in pref[iPersonne]:
      if affect[jMaison] == None:
        affect[jMaison] = iPersonne
        break
  return affect

def reponse4(n):
  pref = [ perm(u, n, 50, i*n) for i in range(n) ]
  ordreAlea = perm(u, n, n, n*n)
  return rang(pref, affectDictSucc(pref, ordreAlea))

print("4.a", reponse4(3))
print("4.b", reponse4(10))
print("4.c", reponse4(100))
print("4.d", reponse4(1000))
```

Question à développer pendant l'oral 1 *Quelle est la complexité en temps de votre implémentation de l'algorithme* `affectDictSucc`, *dans le pire des cas, en fonction de n ?*

Dans l'implémentation ci-dessus, la complexité est de $O(n^2)$. Notez que la mention "dans le pire des cas" invite les candidats à ne surtout pas réfléchir à la complexité en moyenne (les données étant pseudo aléatoires). Il n'est pas demandé de fournir une instance sur laquelle la borne supérieure est atteinte.

3 Algorithme des cycles d'échange

Dans la figure 1, les personnes 0 et 1 aimeraient échanger leurs maisons. Nous allons maintenant étudier un algorithme permettant d'améliorer cette affectation. Dans le problème de ré-affectation des maisons, chaque personne possède initialement une maison différente. Nous souhaitons trouver une affectation dans laquelle chaque personne reçoit une maison qu'elle préfère (ou qui est identique) à la maison qu'elle possédait initialement.

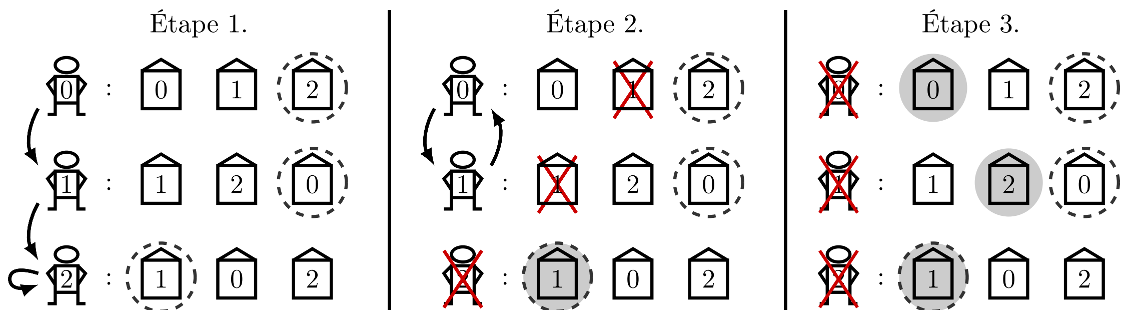
Informellement, l'**algorithme des cycles d'échange** procède de la manière suivante : chaque personne pointe du doigt la personne qui possède sa maison préférée, on calcule un cycle dans le graphe orienté ainsi obtenu, on donne à chaque personne du cycle sa maison préférée, et on recommence sans les maisons et personnes du cycle. Étant donnée une affectation initiale r, on note `affectCycle`(n, r) la ré-affectation s obtenue par l'algorithme des cycles d'échange décrit ci-dessous.

fonction `affectCycle`(n, r)
initialiser un tableau $s \leftarrow [\bot, \ldots, \bot]$ de taille n.
tant que *une des cases de s contient* $\bot$ **:**
définir un graphe G, sur l'ensemble de nœuds $V = \{i : i \notin s\}$.
pour *chaque personne* $i \in V$ **faire**
soit j_i la maison préférée de i, telle que $s[j_i] = \bot$.
ajouter l'arc $i \to r[j_i]$ au graphe G.
fin
calculer un cycle C dans le graphe G.
pour *chaque personne* $i \in C$ **faire**
affecter $s[j_i] \leftarrow i$.
fin
fin
renvoyer s
fin

On remarque qu'à chaque itération, si une maison j n'a pas encore été ré-affectée ($s[j] = \bot$), alors son propriétaire n'a pas encore reçu de maison ($r[j] \in V$). Par conséquent, le graphe G est bien défini.

Question à développer pendant l'oral 2 *Dans le graphe G défini à chaque itération, chaque sommet $i \in V$ a exactement un arc sortant. Démontrer que le graphe G possède au moins un cycle C, et proposez une procédure simple pour le calculer.*

Définissons une séquence qui commence à $i_0 \in V$, telle que i_{k+1} est l'unique successeur de i_k, pour tout $k \geq 0$. Comme il existe un nombre fini de sommets dans V, la séquence se répète forcément. La première répétition nous donne un cycle. Pour calculer le cycle, on itère sur k en maintenant dans un tableau les sommets déjà parcourus.



L'affectation initiale `affectAlea` $= [1, 2, 0]$ est représentée en pointillé. À la fin de l'algorithme, on obtient l'affectation `affectCycle`$(n,$ `affectAlea`$) = [0, 2, 1]$, avec un rang total de $1 = 0 + 1 + 0$.

FIGURE 3 – Algorithme des cycles d'échange, avec $n = 3$ et $\widetilde{u_0} = 489$.

On admettra que la ré-affectation calculée par l'algorithme des cycles d'échange ne dépend pas de l'ordre dans lequel les cycles sont choisis (en effet, si un cycle n'est pas choisi à une itération, il existera toujours à la suivante).

Question 5 *Calculer le rang total de* `affectCycle`$(n,$ `affectAlea`$)$ *avec :*

a) $n = 3$ ***b)*** $n = 10$ ***c)*** $n = 100$ ***d)*** $n = 1000$

```python
def trouveCycle(graphe):
  n = len(graphe)
  vu = [False] * n
  # on trouve un sommet de départ
  for i in range(n):
    if graphe[i] != None:
      vu[i] = True
      seq = [i]
      break
  # on avance jusqu'à retomber sur un sommet déjà vu
  while True:
    suivant = graphe[seq[-1]]
    if vu[suivant]:
      return seq[seq.index(suivant):]
    vu[suivant] = True
    seq.append(suivant)

def affectCycle(pref, initial):
  n = len(pref)
  affect, compteur, actif = [None]*n, [0]*n, [True]*n

  def maison(i): # renvoie la maison (non affectée) préférée de i
    while affect[pref[i][compteur[i]]] != None:
      compteur[i] += 1
    return pref[i][compteur[i]]

  while None in affect:
    graphe = [None] * n
    for iPersonne in range(n):
      if actif[iPersonne]:
        graphe[iPersonne] = initial[maison(iPersonne)]
    cycle = trouveCycle(graphe)
    for iPersonne in cycle:
      affect[maison(iPersonne)] = iPersonne
      actif[iPersonne] = False
  return affect

def reponse5(n):
  pref = [ perm(u, n, 50, i*n) for i in range(n) ]
  affectAlea = perm(u, n, n, n*n)
  return rang(pref, affectCycle(pref, affectAlea))

print("5.a", reponse5(3))
print("5.b", reponse5(10))
print("5.c", reponse5(100))
print("5.d", reponse5(1000))
```

Question à développer pendant l'oral 3 *Quelle est la complexité en temps de votre implémen-*

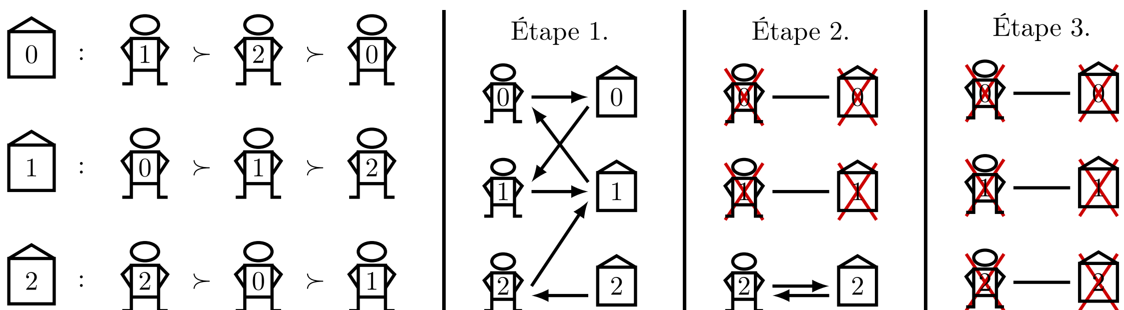
tation de l'algorithme `affectCycle`*, dans le pire des cas, en fonction de n ?*

Un implémentation "naïve" a une complexité en $O(n^3)$. L'implémentation ci-dessus a une la complexité en $O(n^2)$. Pour avoir un algorithme quadratique, il faut remarquer que :
- À chaque itération, au moins une personne reçoit une maison, il y a donc au plus n itérations. Chaque recherche de cycle prend un temps $O(n)$, soit au $O(n^2)$ au total.
- Attention, chercher en temps $O(n)$ la maison (non-affectée) préférée de chaque personne donnerait une complexité en $O(n^3)$ au total. On remarque que à l'aide d'un compteur pour chaque personne, on peut recommencer la recherche là où elle s'était arrêtée à l'itération précédente, ce qui donne une complexité amortie en $O(n^2)$.

4 Algorithme des cycles d'échange généralisé

Dans cette quatrième partie, nous allons étudier le problème d'affectation lorsque chaque maison a un ordre de priorité. Plus précisément, chaque maison j possède une permutation p_j, où $p_j[0]$ est la personne qui a la plus haute priorité, $p_j[1]$ a la seconde priorité, ..., et $p_j[n-1]$ a la plus faible priorité.

Nous **généralisons l'algorithme des cycles d'échange** de la manière suivante : chaque personne pointe du doigt sa maison préférée, chaque maison pointe du doigt sa personne préférée, on calcule un cycle dans le graphe orienté ainsi obtenu, on donne à chaque personne du cycle sa maison préférée, et on recommence sans les maisons et personnes du cycle. On note `affectCyclePrio`$(n, p_0, \ldots, p_{n-1})$ l'affectation obtenue par l'algorithme des cycles d'échange généralisé.



On obtient `affectCyclePrio`$(3, p_0, p_1, p_2) = [0, 1, 2]$, avec un rang total de $2 = 0 + 0 + 2$.

FIGURE 4 – Algorithme des cycles d'échange généralisé, avec $n = 3$ et $\widetilde{u_0} = 489$.

Question à développer pendant l'oral 4 *Quelle affectation obtient-on si toutes les maisons ont le même ordre de priorité ? Et si toutes les maisons ont des premiers choix distincts ?*

À nouveau, on admettra que la ré-affectation calculée par l'algorithme des cycles d'échange généralisé ne dépend pas de l'ordre dans lequel les cycles sont choisis.

Si toutes les maison ont le même ordre de priorité p, alors l'affectation obtenue est celle renvoyée par l'algorithme des dictateurs successifs en utilisant l'ordre p. En effet, la personne $p[0]$ formera un cycle avec sa maison préférée, puis, la personne $p[1]$ formera un cycle avec sa maison restante préférée, etc.
Si chaque maison i à un premier choix $p[i]$ différent, alors l'affectation obtenue est celle renvoyée par l'algorithme des cycles d'échange sur l'affectation initiale p. En effet, chaque maison i pointera toujours sur son "propriétaire" $p[i]$ jusqu'à ce qu'elle fasse parti d'un cycle.

Question 6 *Pour chaque maison j, on construit l'ordre de priorité $p_j = \texttt{perm}(n, n, n^2 + n \cdot j)$. Calculer le rang total de* `affectCyclePrio`$(n, p_0, \ldots, p_{n-1})$ *avec :*

a) $n = 3$ ***b)*** $n = 10$ ***c)*** $n = 100$ ***d)*** $n = 1000$

```
def affectCyclePrio(pref, prio):
  n = len(pref)
  affect, compteur, actif = [None]*n, [[0]* n, [0]*n], [True]*n

  def maison(i): # renvoie la maison (non affectée) préférée de i
    while affect[pref[i][compteur[0][i]]] != None: # déjà affectée
      compteur[0][i] += 1
    return pref[i][compteur[0][i]]

  def personne(j): # renvoie la personne (non affectée) préférée de j
    while not actif[prio[j][compteur[1][j]]]: # déjà affectée
      compteur[1][j] += 1
    return prio[j][compteur[1][j]]

  while None in affect:
    graphe = [None] * n
    for iPersonne in range(n):
      if actif[iPersonne]:
        graphe[iPersonne] = personne(maison(iPersonne))
    cycle = trouveCycle(graphe)
    for iPersonne in cycle:
      affect[maison(iPersonne)] = iPersonne
      actif[iPersonne] = False
  return affect

def reponse6(n):
  pref = [ perm(u, n, 50, i*n) for i in range(n) ]
  prio = [ perm(u, n, n, i*n+n**2) for i in range(n) ]
  return rang(pref, affectCyclePrio(pref, prio))

print("6.a", reponse6(3))
print("6.b", reponse6(10))
print("6.c", reponse6(100))
print("6.d", reponse6(1000))
```

5 Algorithme d'acceptation différée

Dans la figure 4, la personne 2 aurait préféré avoir la maison 0, dans laquelle elle a une priorité plus élevée que la personne 0. Nous allons essayer de résoudre ce problème avec l'algorithme d'acceptation différée.

Informellement, l'**algorithme d'affectation différée** procède de la manière suivante. À chaque itération, on choisit une personne qui n'a pas encore de maison. On la laisse choisir sa maison préférée parmi les maisons non-affectées, et les maisons où elle a une priorité plus élevée que le propriétaire actuel. L'algorithme se termine lorsque chaque personne a une maison. Plus formellement, on note `affectAccDiff`$(n, p_0, \ldots, p_{n-1})$ l'affectation s obtenue par l'algorithme d'acceptation différée décrit ci-dessous.

fonction `affectAccDiff`$(n, p_0, \ldots, p_{n-1})$
  initialiser un tableau $s \leftarrow [\perp, \ldots, \perp]$ de taille n.
  tant que *il existe une personne* $i \notin s$ **:**
    soit j la maison préférée de i, telle que :
      $s[j] = \perp$ ou i est mieux classé que $s[j]$ dans p_j.
    affecter $s[j] \leftarrow i$.
  fin
  renvoyer s
fin

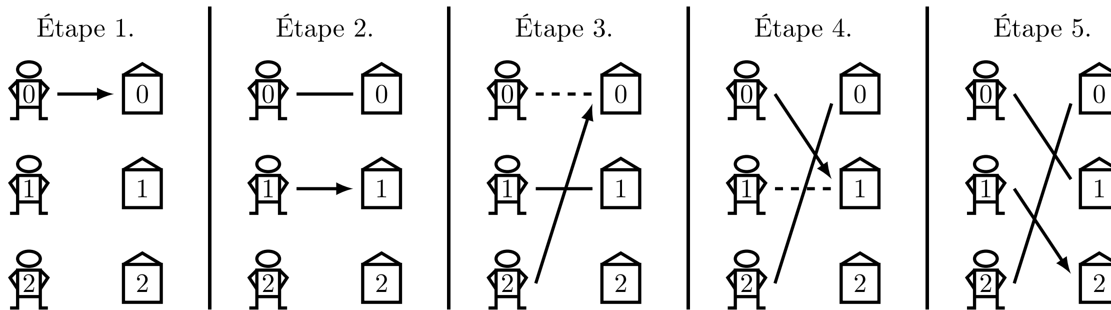
Question à développer pendant l'oral 5 *Donner une borne supérieure sur le nombre d'itérations de la boucle principale.*

À chaque itération, soit une nouvelle maison est affectée, soit une maison est affectée à un propriétaire avec une priorité plus haute. Chaque maison sera affectée une fois et réaffectée au plus n fois, ce qui donne une borne de n^2 sur le nombre d'itérations de la boucle principale.

On admettra que l'affectation calculée par l'algorithme ne dépend pas de l'ordre dans lequel on choisit les personnes à affecter. Dans notre implémentation, nous choisirons à chaque itération la personne $i \notin s$ avec l'indice le plus petit.

Question 7 *Pour chaque maison j, on construit l'ordre de priorité $p_j =$ `perm`$(n, n, n^2 + n \cdot j)$. Calculer le rang total de* `affectAccDiff`$(n, p_0, \ldots, p_{n-1})$ *avec :*

a) $n = 3$ ***b)*** $n = 10$ ***c)*** $n = 100$ ***d)*** $n = 1000$



À chaque étape, la flèche représente le choix de i et j, et une ligne pointillée représente l'ancienne affectation de la maison j. On obtient `affectAccDiff`$(3, p_0, p_1, p_2) = [2, 0, 1]$, avec un rang total de $3 = 1 + 1 + 1$.

FIGURE 5 – Algorithme d'acceptation différée, avec $n = 3$ et $\widetilde{u_0} = 489$.

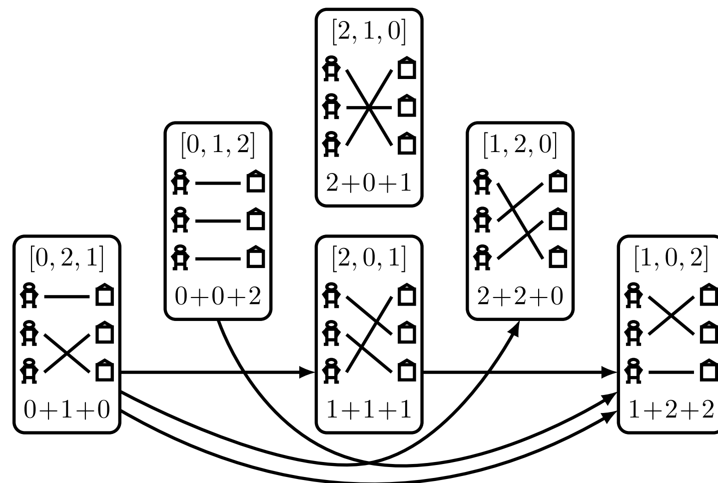
```
def inv(t):
  res = [None] * len(t)
  for i,x in enumerate(t):
    res[x] = i
  return res

def affectAccDiff(pref, prio):
  n = len(pref)
  rang = [inv(p) for p in prio]
  affect, compteur, iPersonne, nb = [None] * n, [0]* n, 0, 0
  while nb < n:
    jMaison = pref[iPersonne][compteur[iPersonne]]
    if affect[jMaison] == None:
      affect[jMaison], iPersonne, nb = iPersonne, nb+1, nb+1
    elif rang[jMaison][iPersonne] > rang[jMaison][affect[jMaison]]:
      compteur[iPersonne] += 1
    else:
      iPersonne, affect[jMaison] = affect[jMaison], iPersonne
  return affect

def reponse7(n):
  pref = [ perm(u, n, 50, i*n) for i in range(n) ]
  prio = [ perm(u, n, n, i*n+n**2) for i in range(n) ]
  return rang(pref, affectAccDiff(pref, prio))

print("7.a", reponse7(3))
print("7.b", reponse7(10))
print("7.c", reponse7(100))
print("7.d", reponse7(1000))
```

Question à développer pendant l'oral 6 *Quelle est la complexité en temps de votre implémentation de l'algorithme* `affectAccDiff`*, dans le pire des cas, en fonction de n ?*

L'implémentation ci-dessus a une complexité de $O(n^2)$, et il existe des implémentations "naïve" avec des complexité en $O(n^3)$ ou $O(n^4)$. Pour avoir un algorithme quadratique, il faut faire attention à ce que :
— le choix de $i \notin s$ est fait en temps constant, en maintenant un compteur du nombre de personnes affectées,
— le choix de j est fait en temps constant **amorti** : pour chaque personne i, on incrémente `compteur`$[i]$ tel que les `compteur`$[i]$ maisons préférées de i sont affectées à des personnes ayant une plus haute priorité,
— la comparaison entre i et $s[j]$ est faite en temps constant : on pré-calcule le rang de chaque personne dans chaque ordre de priorité.

6 Optimum de Pareto

On dit qu'une affectation s' **améliore** une affectation s, et on notera $s' \succeq s$, si chaque personne reçoit dans s' une maison qu'elle préfère (ou qui est identique) à celle qu'elle reçoit dans s. Étant donné une affectation s, on dira que s est **améliorable** s'il existe $s' \neq s$ tel que $s' \succeq s$, et sinon on dira que s est un **optimum** (de Pareto).



Sous chaque affectation est renseigné son rang total. Un arc entre deux affectations $s' \to s$ signifie que s' améliore s. Il existe trois optima (affectations sans arc entrant).

FIGURE 6 – Toutes les affectation possibles, avec $n = 3$ et $\widetilde{u_0} = 489$.

Nous allons admettre que les propriétés suivantes sont vérifiées :
— l'algorithme des dictateurs successifs renvoie toujours un optimum ;
— tout optimum peut être obtenu avec l'algorithme des cycles d'échange (avec un choix astucieux d'affectation initiale) ;
— toute affectation obtenue par l'algorithme des cycles d'échange peut aussi être obtenue par l'algorithme des dictateurs successifs (avec un choix astucieux d'ordre).

Ces trois propriétés nous permettent de caractériser l'ensemble des optima. En utilisant cette caractérisation, nous souhaitons calculer le nombre d'optima.

Question 8 *Calculer le nombre d'optima différents, avec :*

a) $n = 3$ ***b)*** $n = 5$ ***c)*** $n = 8$

d) $n = 10$ ***e)*** $n = 12$ ***f)*** $n = 15$

```python
def enumerate_pareto(pref):
  n = len(pref)
  def rec(s, b):
    # si s[j] == i, la maison j est déjà affectée à i.
    # si b[i] == j, la maison j ne devra jamais être affectée à i.
    if None not in s:
      yield tuple(s)
      return
    for i in range(n):
      if i not in s:
        # i n'est pas encore affectée
        if b[i] == None or s[b[i]] != None:
          # on peut donner à i sa maison (non affectée) préférée
          for j in pref[i]:
            if s[j] == None:
              # j est la maison (non affectée) préférée de i.
              s2, b2 = s.copy(), b.copy()
              s2[j], b2[i] = i, j
              yield from rec(s2, b.copy()) # on affecte j à i
              yield from rec(s.copy(), b2) # on n'affecte pas j à i
              return
  yield from rec([None]*n, [None]*n)

def reponse8(n):
  pref = [ perm(u, n, 50, i*n) for i in range(n) ]
  return len(list(enumerate_pareto(pref)))

print("8.a", reponse8(3))
print("8.b", reponse8(5))
print("8.c", reponse8(8))
print("8.d", reponse8(10))
print("8.e", reponse8(12))
print("8.f", reponse8(15))
```

Question à développer pendant l'oral 7 *Expliquer votre algorithme pour compter le nombre d'optima, ainsi que les éventuelles optimisations améliorant le temps de calcul.*

On remarque qu'une affectation est un optimum si et seulement si elle peut être obtenue avec l'algorithme des dictateurs successifs. On peut tester tous les ordres possibles en éliminant les doublons parmi les affectations obtenues, ce permet de calculer la réponse pour $n \leq 8$ en stockant les affectation obtenues dans une liste, et pour $n \leq 10$ en stockant les affectations obtenues dans une table de hachage.

Pour calculer la réponse pour $n \leq 15$, il faut implémenter une fonction récursive, qui énumère les exécutions possibles de l'algorithme des dictateurs successifs, en évitant de générer des doublons. Dans l'implémentation ci-dessus, le tableau `interdit` permet d'éviter de générer une affectation qui sera obtenu par un autre appel récursif.

7 Stabilité

Dans le problème d'affectation avec des priorités (sections 4 et 5), on dit qu'une personne i et une maison j bloquent une affectation s si i et j se préfèrent mutuellement à leurs affectations respectives. On dit qu'une affectation est **stable** si elle n'a pas de couple (i, j) bloquant.

Dans la figure 4, la personne $i = 2$ et la maison $j = 0$ bloquent l'affectation (car i préfère j à la maison 2, et j préfère i à la personne 0), qui n'est donc pas stable. Dans la figure 5, aucun couple ne bloque l'affectation, qui est donc stable. On admettra que l'algorithme d'affectation différée renvoie toujours une affectation stable.

Dans l'exemple avec $n = 3$ et $\widetilde{u_0} = 489$, il existe deux affectations stables : $[2, 0, 1]$ renvoyée par `affectAccDiff`, et $[1, 0, 2]$. Nous souhaitons calculer le nombre d'affectations stables.

Question 9 *Pour chaque maison j, on construit l'ordre de priorité $p_j =$ `perm`$(n, n, n^2 + n \cdot j)$. Calculer le nombre d'affectation stables, avec :*

a) $n = 3$ ***b)*** $n = 10$ ***c)*** $n = 15$

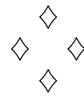
d) $n = 20$ ***e)*** $n = 30$ ***f)*** $n = 50$

```python
def enumerate_stable(pref, prio):
  n = len(pref)
  A = [inv(p) for p in pref]
  B = [inv(p) for p in prio]
  def rec(a, b, s):
    if None not in s:
      yield tuple(s)
      return
    # i doit être affectée une des maison de pref[i][:a[i]]
    # j doit être affectée à une des personnes de prio[j][:b[j]]
    # essayons toutes les affectation possible d'une personne/maison
    # on choisit cette personne/maison en minimisant le nb de branchements
    tbest = [None]*(n+1)
    for i in range(n):
      if a[i] != -1: # pas encore affectée
        t = [(i,j) for j in pref[i][:a[i]] if B[j][i] < b[j]]
        if len(t) < len(tbest): tbest = t
    for j in range(n):
      if b[j] != -1: # pas encore affectée
        t = [(i,j) for i in prio[j][:b[j]] if A[i][j] < a[i]]
        if len(t) < len(tbest): tbest = t
    for i,j in tbest:
      a2, b2, s2 = a.copy(), b.copy(), s.copy()
      a2[i], b2[j], s2[j] = -1, -1, i
      for i2 in prio[j][:B[j][i]]:
        a2[i2] = min(a2[i2], A[i2][j])
      for j2 in pref[i][:A[i][j]]:
        b2[j2] = min(b2[j2], B[j2][i])
      yield from rec(a2, b2, s2)
  yield from rec([n]*n, [n]*n, [None]*n)

def reponse9(n):
  pref = [ perm(u, n, 50, i*n) for i in range(n) ]
  prio = [ perm(u, n, n, i*n+n**2) for i in range(n) ]
  return len(list(enumerate_stable(pref, prio)))

print("9.a", reponse9(3))
print("9.b", reponse9(10))
print("9.c", reponse9(15))
print("9.d", reponse9(20))
print("9.e", reponse9(30))
print("9.f", reponse9(50))
```

Question à développer pendant l'oral 8 *Expliquer votre algorithme pour compter le nombre d'affectations stables, ainsi que les éventuelles optimisations améliorant le temps de calcul.*

Il est possible de générer toutes les affectations possibles, par exemple avec une fonction récursive, puis de vérifier quelles affectations sont stables, ce qui permet de calculer la réponse pour $n \leq 10$.

Une première optimisation dans l'algorithme récursif, consiste à arrêter d'explorer une branche dès qu'une instabilité est détectée, ce qui permet de calculer la réponse pour $n \leq 15$.
Une deuxième optimisation consiste à améliorer les vérifications d'instabilité, en maintenant pour chaque personne/maison le rang maximal d'une affectation stable (variables `a` et `b` dans le code ci-dessus). Cela permet de calculer la réponse pour tout $n \leq 20$.
La troisième optimisation consiste à minimiser le nombre de branchements dans la fonction récursive. À chaque appel, la fonction récursive essaie d'affecter toutes les maisons possibles à une personne fixée. Fixer la personne qui peut recevoir le moins de maison est une heuristique qui permet de calculer la réponse pour tout $n \leq 30$. Observer la symétrie du problème et fixer la meilleur personne ou maison à affecter permet de calculer la réponse pour tout $n \leq 50$.



Fiche réponse type : Affectation de maisons

$\widetilde{u_0}$: 489

Question 1

a) 71 502

b) 88 966

c) 82 070

d) 18 545

Question 2

a) [0, 2, 1]

b) [1, 4, 3, 0, 2]

c) [1, 4, 3, 7, 6, 5, 0, 2]

d) [1, 4, 3, 7, 6, 9, 0, 2, 8, 5]

Question 3

a) 4

b) 51

c) 4537

d) 499 406

Question 4

a) 3

b) 17

c) 2878

d) 478 941

Question 5

a) 1

b) 19

c) 2922

d) 478 674

Question 6

a) 2

b) 16

c) 2902

d) 478 628

Question 7

a) 3

b) 19

c) 3368

d) 496 738

Question 8

a) 3

b) 9

c) 1034

d) 3325

e) 8434

f) 130 529

Question 9

a) 2

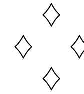
b) 2

c) 4

d) 11

e) 8

f) 20



Calculs et tableurs

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

Durée de l'épreuve : 3 heures 30 minutes

Juin 2024

ATTENTION !

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

Important.

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

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

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

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

1 Préliminaires

Les sections 3, 4, 5 et 6 peuvent être traitées de manière indépendante mais il est néanmoins recommandé de les traiter dans l'ordre.

1.1 Notations

Pour deux entiers naturels a et b avec $b \neq 0$, on notera $a \mod b$ le reste de la division euclidienne de a par b, c'est-à-dire l'unique entier r avec $0 \leq r < b$ tel que $a = k \times b + r$ et $k \in \mathbb{N}$.
Attention : En OCaml, le résultat de la fonction `mod` peut être négatif si a est lui-même négatif.

On rappelle que $\mathbb{Z}/k\mathbb{Z}$ est l'anneau des entiers modulo k. Un élément de $\mathbb{Z}/k\mathbb{Z}$ sera en général représenté par un entier $0 \leq x < k$.

On parlera de booléen pour indiquer le type de données contenant vrai et faux.

On note $A \wedge B$ la conjonction des deux formules A et B, et $A \vee B$ leur disjonction.

1.2 Génération de nombres pseudo-aléatoires

Étant donné u_0, on considère la suite u générée par la relation de récurrence suivante :

$$u_{n+1} = (178\,481 \times u_n + 131\,071) \mod 869\,753$$

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. Il est donc conseillé que votre implémentation calcule le tableau de tous les u_n jusqu'à un certain rang choisi soigneusement.
On pourra par ailleurs remarquer que la suite u est cyclique de longueur 869 752.

Question 1 *Calculez les valeurs suivantes :*

a) $u_1 \mod 1000$ ***b)*** $u_{24} \mod 1000$

c) $u_{7852} \mod 1000$ ***d)*** $u_{869\,751} \mod 1000$

```
let unmod = 869_753
let usize = unmod - 1

let next_u x = (178_481 * x + 131_071) mod unmod

let uns =
  let u = Array.make usize 0 in

  u.(0) <- u0 ; (* le u0 donné *)

  for i = 0 to usize - 1 do
    u.(i+1) <- next_u u.(i)
  done ;

  u

(* Pour accéder à u_n il suffit de regarder [n mod usize] dans [uns] *)
let safe_un n =
  uns.(n mod usize)
```

1.3 Tableurs

Les tableurs sont des outils logiciels qui contiennent des feuilles de calcul et qui ont des utilisations variées telles que la comptabilité, les statistiques, etc.

Dans ce sujet, on considère une version simplifiée des tableurs. Un tableur va être vu comme une matrice composée de cellules contenant chacune soit un scalaire pris dans l'anneau $\mathbb{Z}/p\mathbb{Z}$ pour un certain p, soit une formule faisant référence à d'autres cellules du tableur. Nous donnons un exemple de tel tableur avec $p = 11$ en Figure 1.

	0	1	2	3	4
0	3	8	PLUS$((0,0),(2,1))$	1	PROD$((2,1),(4,2))$
1	2	MOINS$((0,2))$	9	5	PROD$((3,2),(0,1))$
2	10	PLUS$((1,1),(0,2))$	0	MOINS$((2,0))$	7

FIGURE 1 – Exemple de tableur

Ici le tableur est de taille 5×3 avec des colonnes numérotées de 0 à 4 et des lignes de 0 à 2. La notation (i, j) fait référence à la cellule située colonne i et ligne j. Par exemple, le contenu de la cellule $(3, 2)$ est MOINS$((2, 0))$ qui est une formule faisant référence à la cellule $(2, 0)$. Pour évaluer cette formule, il faut donc évaluer la cellule $(2, 0)$ et prendre son opposé modulo p. La cellule contient elle-même une autre formule : PLUS$((0, 0), (2, 1))$. Celle-ci fait référence aux cellules $(0, 0)$ et $(2, 1)$ qui contiennent cette fois-ci les valeurs 3 et 9 respectivement, ce qui donne comme résultat $3 + 9 = 12 = 1$ pour la cellule $(2, 0)$ et donc $-1 = 10$ pour $(3, 2)$. De même, on évalue une formule PROD(c_1, c_2) en faisant le produit (modulo p) des valeurs correspondant à c_1 et c_2.
Évaluer toutes les cases d'un tableur – quand c'est possible – donne lieu à une vue qui est une matrice de même taille ne contenant que des scalaires. La vue associée au tableur de la Figure 1

est donnée en Figure 2.

	0	1	2	3	4
0	3	8	1	1	8
1	2	1	9	5	9
2	10	0	0	10	7

FIGURE 2 – Vue d'un tableur

1.4 Génération de formules et cellules

Dans ce sujet on va considérer que les formules ne sont jamais imbriquées et sont donc toujours de la forme $\mathsf{PLUS}(c_1, c_2)$, $\mathsf{PROD}(c_1, c_2)$ ou $\mathsf{MOINS}(c)$ où c, c_1 et c_2 font référence à des cellules. Ces références sont donc de la forme (i, j). Le contenu d'une cellule est soit une formule, soit un entier.

Afin de générer formules et cellules, on définit deux fonctions, v et w par :

$$v(m,i,j) = u_{3(im+j)+1} \mod m \qquad\qquad w(m,i,j) = u_{3(im+j)+2} \mod m$$

On définit une fonction $\mathfrak{c}$ qui renvoie le contenu d'une cellule (donc soit une formule soit un entier). $\mathfrak{c}$ prend pour arguments un booléen b et cinq entiers n, m, p, i, j et est définie comme suit :

$$\mathfrak{c}(b,n,m,p,i,j) = \begin{cases} \mathsf{PLUS}((i-1, v(m,i,j)), (i-1, w(m,i,j))) & \text{si} \quad r = 0 \\ \mathsf{PROD}((i-1, v(m,i,j)), (i-1, w(m,i,j))) & \text{si} \quad r = 1 \\ \mathsf{MOINS}((i-1, v(m,i,j))) & \text{si} \quad (r = 2) \lor ((r = 3) \land \neg b') \\ \mathsf{MOINS}((n-1, v(m,i,j))) & \text{si} \quad (r = 3) \land b' \\ u_{3(im+j)+1} \mod p & \text{sinon} \end{cases}$$

où $r = u_{3(im+j)} \mod 6$ et $b' = b \land (u_{3(im+j)+2} \mod 17 = 0)$.

On définit également la notion de somme de contrôle $\mathsf{ccs}(c)$ d'une cellule c, en utilisant la somme de contrôle pour une référence rcs.

$$\mathsf{rcs}((i,j)) = i^3 + j, \qquad \mathsf{ccs}(\mathsf{PLUS}(c_1,c_2)) = \mathsf{rcs}(c_1) + \mathsf{rcs}(c_2),$$

$$\mathsf{ccs}(\mathsf{PROD}(c_1,c_2)) = \mathsf{rcs}(c_1)^2 + \mathsf{rcs}(c_2), \qquad \mathsf{ccs}(\mathsf{MOINS}(c)) = 1 + \mathsf{rcs}(c), \qquad \mathsf{ccs}(i) = i.$$

Question 2 *Calculez la somme (modulo 10 000) de contrôle* ccs *des cellules suivantes :*

a) $\mathfrak{c}(\mathsf{vrai}, 10, 10, 97, 5, 5)$

b) $\mathfrak{c}(\mathsf{vrai}, 10, 100, 547, 3, 97)$

c) $\mathfrak{c}(\mathsf{vrai}, 30, 1000, 1091, 23, 502)$

d) $\mathfrak{c}(\mathsf{vrai}, 1000, 2000, 5099, 297, 150)$

```ocaml
(* Types de données pour représenter les cellules *)

type cref = int * int

type formula =
| PLUS of cref * cref
| TIMES of cref * cref
| MINUS of cref

type cell =
| Value of int
| Formula of formula

(* Génération de cellule *)

let gen_cell b n m p i j =
  let op = safe_un (3 * (i * m + j)) mod 6 in
  let r1 = safe_un (3 * (i * m + j) + 1) in
  let r2 = safe_un (3 * (i * m + j) + 2) in
  let c1 = (i-1, r1 mod m) in
  let c2 = (i-1, r2 mod m) in
  if op = 0 then Formula (PLUS (c1,c2))
  else if op = 1 then Formula (TIMES (c1,c2))
  else if op = 2 then Formula (MINUS c1)
  else if op = 3 then (
    if b && r2 mod 17 = 0 then Formula (MINUS (n-1, r1 mod m))
    else Formula (MINUS c1)
  )
  else Value (r1 mod p)

(* Somme de contrôle *)

let cref_checksum (i,j) =
  i * i * i + j

let rec formula_checksum f =
  match f with
  | PLUS (c1,c2) -> cref_checksum c1 + cref_checksum c2
  | TIMES (c1,c2) -> cref_checksum c1 * cref_checksum c1 + cref_checksum c2
  | MINUS c -> 1 + cref_checksum c

let cell_checksum c =
  match c with
  | Value i -> i mod 10_000
  | Formula f -> formula_checksum f mod 10_000
```

1.5 Génération de tableurs

Comme indiqué précédemment un tableur est donné par ses dimensions $n \times m$ (n colonnes et m lignes), le modulo p, et par le contenu (formule ou valeur) de ses $n \times m$ cellules.

On définit une fonction T telle que $\mathsf{T}(b, n, m, p)$ génère un tableur de dimension $n \times m$ et où le

contenu de ses cellules (modulo p) est donné par

$$\begin{array}{rcll} \mathsf{T}(b,n,m,p)(0,j) & = & u_j \ \mathsf{mod}\ p & \text{pour } 0 \leq j < m \\ \mathsf{T}(b,n,m,p)(i,j) & = & \mathfrak{c}(b,n,m,p,i,j) & \text{pour } 0 < i < n \wedge 0 \leq j < m. \end{array}$$

On définit la somme de contrôle $\mathsf{tcs}(T)$ d'un tableur T comme suit :

$$\mathsf{tcs}(T) = (\sum_{i=0}^{n-1} \sum_{j=0}^{m-1} \mathsf{ccs}(T(i,j))) \ \mathsf{mod}\ 10\,000$$

Question 3 *Calculez la somme de contrôle* tcs *des tableurs suivants :*

a) $\mathsf{T}(\mathsf{vrai}, 10, 10, 97)$

b) $\mathsf{T}(\mathsf{faux}, 10, 100, 547)$

c) $\mathsf{T}(\mathsf{faux}, 30, 1000, 1091)$

d) $\mathsf{T}(\mathsf{vrai}, 1000, 2000, 5099)$

```
(* Représentation d'un tableur *)
type spreadsheet = cell array array

(* Génération *)
let gen_spreadsheet b n m p : spreadsheet =
  let sheet = Array.make_matrix n m (Value 0) in
  for j = 0 to m - 1 do
    sheet.(0).(j) <- Value ((safe_un j) mod p)
  done ;
  for i = 1 to n - 1 do
    for j = 0 to m - 1 do
      sheet.(i).(j) <- gen_cell b n m p i j
    done
  done ;
  sheet

(* Somme de contrôle *)
let spreadsheet_checksum sheet =
  let n = Array.length sheet in
  let m = Array.length sheet.(0) in
  let sum = ref 0 in
  for i = 0 to n - 1 do
    for j = 0 to m - 1 do
      sum := (!sum + cell_checksum sheet.(i).(j)) mod 10_000
    done
  done ;
  !sum
```

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

Il s'agit d'une question très simple : la complexité en temps de la somme de contrôle est linéaire en le nombre de cellules, soit en $\mathcal{O}(n \times m)$.
Pour ce qui est de la génération des données, nous faisons le choix de les stocker dans un tableau, qui prend un espace mémoire et un temps d'initialisation également en $\mathcal{O}(n \times m)$. On aurait cependant pu s'en passer en faisant appel à $\mathfrak{c}$ directement.

2 Graphes de dépendances

Dans un tableur, chaque cellule peut faire référence à 0, 1 ou 2 autres cellules. On peut construire un graphe orienté qui représente ces références en ajoutant un arc de c_1 à c_2 quand la cellule c_2 fait référence à c_1, par exemple si c_2 contient la formule MOINS(c_1).

Autrement dit, un arc de c_1 à c_2 indique que c_1 doit être évaluée avant c_2, puisque la valeur de c_2 dépend de celle de c_1. On peut remarquer dans ce cas que la présence de cycles dans le graphe de dépendances signifie l'impossibilité d'évaluer le tableur.

Le graphe de dépendances associé au tableur de la Figure 1 est donné en Figure 3. Les nœuds du graphe correspondent aux différentes références à des cellules et les arcs aux dépendances entre celles-ci. Les arcs sortants d'un nœud c sont les arcs allant de c à c' pour tout autre c'. Par exemple, le nœud $(2, 1)$ a deux arcs sortants, un vers $(2, 0)$ et un vers $(4, 0)$. Au total, le graphe contient 8 nœuds avec arcs sortants.

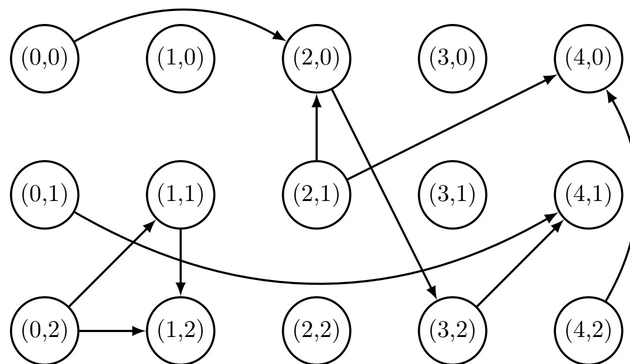


FIGURE 3 – Graphe de dépendances

Question 4 *Calculez le nombre, modulo* 100 000, *de nœuds avec des arcs sortants dans les graphes de dépendances associés aux tableurs suivants :*

a) T(vrai, 10, 10, 97)

b) T(vrai, 10, 100, 547)

c) T(vrai, 30, 1000, 1091)

d) T(vrai, 1000, 2000, 5099)

```
(** Graphe de dépendances

  On le voit comme un tableau qui à chaque cellule associe la liste des
  cellules qui en dépendent.
  (L'autre direction est triviale.)

**)

type dpdgraph = cref list array array

(** Insertion

  Pour éviter les doublons, on exploite le fait que les nœuds sont ajoutés dans
  l'ordre croissant pour les couples (i,j). Autrement dit, quand on ajoute une
  référence à une cellule, si elle est déjà présente dans la liste, c'est
  forcément en tête.

**)

let insert x l =
  match l with
  | [] -> [x]
  | y :: l when x = y -> y :: l
  | _ -> x :: l

(* c dépend de (i,j) *)
let add_dpd (dpd : dpdgraph) (i,j) c =
  dpd.(i).(j) <- insert c dpd.(i).(j)

let make_dpd (sheet : spreadsheet) : dpdgraph =
  let n = Array.length sheet in
  let m = Array.length sheet.(0) in
  let dpd = Array.make_matrix n m [] in
  for i = 1 to n - 1 do
    for j = 0 to m - 1 do
      begin match sheet.(i).(j) with
      | Value i -> ()
      | Formula (PLUS (c1,c2))
      | Formula (TIMES (c1,c2)) ->
        add_dpd dpd c1 (i,j) ;
        add_dpd dpd c2 (i,j)
      | Formula (MINUS c) ->
        add_dpd dpd c (i,j)
      end
    done
  done ;
  dpd
```

Question à développer pendant l'oral 2 *Détaillez quelle structure de données vous avez choisie pour représenter les graphes de dépendances et justifiez votre choix.*

Pour représenter un graphe, on doit souvent faire le choix entre une matrice d'adjacence ou une liste d'adjacence. La matrice d'adjacence est pertinente lorsque le nombre d'arcs est élevé par rapport au nombre de nœuds du graphe. Ce n'est pas le cas ici et nous choisissons donc d'utiliser des listes d'adjacence : à chaque nœud on associe la liste de ses "successeurs". Cela permet également un parcours efficace du graphe comparé aux matrices d'adjacence.

3 Tri topologique et évaluation de tableur

Comme déjà suggéré, le graphe de dépendances, lorsqu'il est acyclique, permet de déterminer un ordre d'évaluation, c'est-à-dire un ordre total sur les cellules qui respecte les dépendances : si une cellule c_2 dépend d'une autre cellule c_1 alors c_1 est "plus petite" que c_2 et doit être évaluée avant. Un tel ordre s'appelle ordre topologique, et il peut en exister plusieurs. Par exemple, dans le graphe en Figure 3, les cellules $(1, 0)$ et $(3, 0)$ peuvent être évaluées dans n'importe quel ordre. On va donc s'intéresser au fait de trouver un ordre topologique sans se préoccuper duquel il s'agit. Pour ce sujet, l'ordre choisi n'a pas d'influence du moment que c'est bien un ordre topologique.

Un tri topologique consiste à trouver un ordre topologique. Il peut se faire en effectuant un parcours en profondeur du graphe de dépendances, tout en prenant soin de distinguer les nœuds déjà visités et intégrés à l'ordre, les nœuds en train d'être visités, et les nœuds pas encore visités.

Un tel algorithme va donc détecter les cycles du graphe. On donne un exemple de tableur avec une référence circulaire en Figure 4 et le graphe de dépendances associé en Figure 5. Remarquez la présence d'un cycle. Celui-ci empêche d'obtenir une vue du tableur car il est impossible d'évaluer la cellule $(0, 0)$ par exemple.

	0	1	2	3
0	MOINS$((2, 0))$	4	PROD$((3, 0), (3, 1))$	10
1	5	PLUS$((0, 0), (1, 0))$	0	MOINS$((1, 1))$

FIGURE 4 – Exemple de tableur avec une référence circulaire

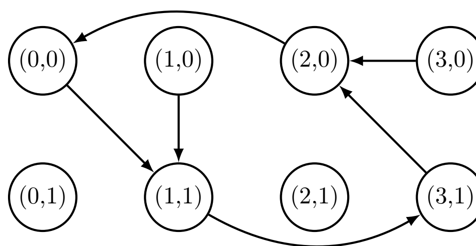


FIGURE 5 – Graphe de dépendances avec cycle

Si un ordre topologique peut être établi (donc s'il n'y a pas de cycles), on peut alors calculer une vue du tableur associé en évaluant les formules dans cet ordre.

Question 5 *Calculez la somme modulo p des valeurs des cellules dans la vue associée au tableur* $\mathsf{T}(\mathsf{vrai}, n, m, p)$ *pour les valeurs de n, m et p suivantes. Si le tableur présente un cycle, indiquez simplement "Cycle" dans la fiche réponse.*

- ***a)*** $n = 10,\ m = 10,\ p = 97$
- ***b)*** $n = 10,\ m = 100,\ p = 547$
- ***c)*** $n = 30,\ m = 1000,\ p = 1091$
- ***d)*** $n = 1000,\ m = 2000,\ p = 5099$

```
(* Type des marques, pour indiquer... *)
type marker =
| Blank (* ... les cases non visitées, *)
| Ongoing (* ... les cases en cours de visite, *)
| Done (* ... les cases visitées. *)

(* Exception à lever en cas de cycle *)
exception Cycle_found

let toposort (dpd : dpdgraph) : cref list option =
  let n = Array.length dpd in
  let m = Array.length dpd.(0) in
  let mark = Array.make_matrix n m Blank in
  let ord = ref [] in

  let rec visit (i,j) =
    match mark.(i).(j) with
    | Done -> ()
    | Ongoing -> raise Cycle_found
    | Blank ->
      mark.(i).(j) <- Ongoing ;
      List.iter visit dpd.(i).(j) ;
      mark.(i).(j) <- Done ;
      ord := (i,j) :: !ord
  in

  try begin
    for i = 0 to n - 1 do
      for j = 0 to m - 1 do
        visit (i,j)
      done
    done ;
    Some !ord
  end
  with Cycle_found -> None
```

```
(* Type des vues *)
type spreadview = int array array

let eval_cref (view : spreadview) (i,j) =
  view.(i).(j)

let eval_formula p view f =
  match f with
  | PLUS (c1,c2) -> (eval_cref view c1 + eval_cref view c2) mod p
  | TIMES (c1,c2) -> (eval_cref view c1 * eval_cref view c2) mod p
  | MINUS c -> p - eval_cref view c

let eval_cell p (sheet : spreadsheet) view i j =
  match sheet.(i).(j) with
  | Value i -> i
  | Formula f -> eval_formula p view f

let topo_eval_sheet p sheet (ord : cref list) : spreadview =
  let n = Array.length sheet in
  let m = Array.length sheet.(0) in
  let view = Array.make_matrix n m 0 in
  List.iter (fun (i,j) ->
    view.(i).(j) <- eval_cell p sheet view i j
  ) ord ;
  view

let view_checksum p (view : spreadview) =
  let n = Array.length view in
  let m = Array.length view.(0) in
  let sum = ref 0 in
  for i = 0 to n - 1 do
    for j = 0 to m - 1 do
      sum := (!sum + view.(i).(j)) mod p
    done
  done ;
  !sum
```

Question à développer pendant l'oral 3 *Décrivez l'algorithme utilisé pour établir l'ordre topologique en insistant sur les structures de données utilisées. Donnez une analyse de la complexité en fonction de n et m, les dimensions du tableur.*

Pour le tri topologique on va maintenir un tableau contenant pour chaque cellule un état indiquant si elle a déjà été visitée ou si elle en cours de visite.
On construit l'ordre topologique au fur et à mesure sous la forme d'une liste de références vers des cellules.
Notre algorithme récursif opère en partant d'une cellule donnée, en la marquant comme "en cours de visite", puis en rendant visite à ses successeurs pour le graphe de dépendances. Si une cellule visitée est marquée comme ayant déjà visité, on ne fait rien ; si elle est "en cours" alors c'est qu'il

y a un cycle ; enfin si elle n'est pas encore marquée on procède récursivement. Après le parcours, la cellule est marquée comme déjà visitée et est ajoutée en tête de la liste de cellules. On s'assure de cette manière que la cellule est située avant tous ses successeurs dans l'ordre.
On procède ainsi tant qu'il existe des cellules non marquées.
Le tri topologique à une complexité linéaire en le nombre de nœuds et d'arcs du graphe. Chaque nœud ayant au plus deux arcs entrants, cela donne au total un temps en $\mathcal{O}(n \times m)$.

Question à développer pendant l'oral 4 *Même question pour l'obtention de la somme des valeurs de la vue.*

Pour l'évaluation, on maintient une matrice contenant les valeurs déjà connues de la vue, en les initialisant d'abord toutes à 0. Pour chaque cellule, prises dans l'ordre topologique, on va évaluer son contenu en lisant les valeurs dans la vue. Comme on effectue le parcours dans l'ordre topologique, à chaque accès de la vue, on est assuré que les valeurs ont bien été calculées.
Ce parcours est linéaire en le nombre de cellules dans l'ordre topologique, c'est à dire en $\mathcal{O}(n \times m)$.

4 Évaluation incrémentale

À partir de maintenant, nous allons considérer que tous les tableurs considérés ne présentent aucun cycle. Attention ! Cela signifie que nous n'allons pas utiliser les mêmes exemples que précédemment.

4.1 Idée générale

Dans cette section nous allons nous intéresser au problème dit du calcul incrémental. Étant donné un tableur et une vue associée, la question est de recalculer efficacement la vue quand seulement une petite partie des données a changé.

Pour ce faire, on va reposer à nouveau sur le graphe de dépendances. Si on revient au tableur Figure 1, changer la valeur de la cellule $(0,0)$ implique de devoir recalculer la cellule $(2,0)$ qui en dépend directement, mais également les cellules $(3,2)$ et $(4,1)$ qui sont dans la fermeture transitive de la relation de dépendance. En revanche, aucune des autres cases n'a besoin d'être recalculée.

Par ailleurs, si on modifie la cellule $(0,2)$, on peut voir que la cellule $(1,2)$ a besoin d'être recalculée pour deux raisons : d'abord parce qu'elle dépend de $(0,2)$, mais aussi parce qu'elle dépend de $(1,1)$ qui elle-même dépend de $(0,2)$. Il faut donc prendre soin de ne pas recalculer cette valeur deux fois inutilement.

L'algorithme va donc opérer en deux passes : une première passe qui consiste à invalider les cellules qui dépendent de la cellule changée, suivie d'une deuxième qui va réparer la vue en recalculant uniquement les cellules invalidées.

Note. On ne va considérer que des modifications qui remplacent des scalaires par d'autres scalaires, ainsi le graphe de dépendances n'est pas modifié.

4.2 Passe d'invalidation

Comme indiqué précédemment, cette phase consiste à marquer comme invalidées les cellules qui dépendent de la cellule modifiée, et ce transitivement. Pour éviter les calculs inutiles, on va également calculer, pour chaque cellule c, le nombre de cellules invalidées c' avec un arc vers c. On appellera ce nombre le degré d'une cellule. Pour tenir compte des cellules qui ont subit le changement, on considèrera que celles-ci ont pour degré 1.

Par exemple (toujours Figure 1), après avoir changé la valeur de la cellule $(0,2)$, la cellule $(1,1)$ aura pour degré 1, tandis que la cellule $(1,2)$ aura pour degré 2. La cellule $(0,2)$ doit elle aussi avoir 1 pour degré. Toutes les autres cellules ont pour degré 0 et n'ont pas besoin d'être changées.

On note $c \leftarrow v$ pour le changement qui remplace le contenu de la cellule c par la valeur v. Par exemple $(0,2) \leftarrow 6$ remplace le contenu de la cellule $(0,2)$ par la valeur 6. Appliquer cette modification au tableur Figure 1 donne un nouveau tableur montré en Figure 6. Les cellules invalidées sont surlignées, avec une emphase particulière pour la cellule directement modifiée.

	0	1	2	3	4
0	3	8	PLUS$((0,0),(2,1))$	1	PROD$((2,1),(4,2))$
1	2	MOINS$((0,2))$	9	5	PROD$((3,2),(0,1))$
2	6	PLUS$((1,1),(0,2))$	0	MOINS$((2,0))$	7

FIGURE 6 – Tableur modifié

Note. Si l'on venait à modifier à nouveau la cellule $(0,2)$ — par exemple avec le changement $(0,2) \leftarrow 4$ — les degrés restent néanmoins inchangés (ils ne se cumulent pas).

Nous allons maintenant supposer que quelques changements ont été effectués dans le première colonne d'un tableur T. On considère la séquence $C(k,p,m)$ de 10 changements suivante :

$$(0, u_{k+2l} \bmod m) \leftarrow u_{k+2l+1} \bmod p$$

pour $0 \leq l < 10$.

Dans les deux questions qui suivent, on considère uniquement la première passe qui invalide les cellules sans effectuer la passe de réparation.

Question 6 *Calculez la somme d des degrés ainsi que le nombre N de cellules invalidées après avoir effectué la séquence de changements $C(79\,730, p, m)$ sur le tableur $\mathsf{T}(\mathsf{faux}, n, m, p)$ pour les valeurs de n, m et p suivantes. Écrivez le résultat sous la forme du couple (d, N).*

a) $n = 10$, $m = 10$, $p = 97$

b) $n = 10$, $m = 100$, $p = 547$

c) $n = 30$, $m = 1000$, $p = 1091$

d) $n = 1000$, $m = 2000$, $p = 5099$

```ocaml
(* Matrice associant à chaque cellule son degré *)
type status = int array array

(* Initialisation des degrés à 0 *)
let make_degree (sheet : spreadsheet) : status =
  let n = Array.length sheet in
  let m = Array.length sheet.(0) in
  Array.make_matrix n m 0

(* Invalide (i,j) et les cellules qui en dépendent *)
let rec invalidate_deps (dpd : dpdgraph) (deg : status) (i,j) =
  if deg.(i).(j) = 0 then begin
    deg.(i).(j) <- 1 ;
    List.iter (invalidate_deps dpd deg) dpd.(i).(j)
  end
  else deg.(i).(j) <- 1 + deg.(i).(j)

(* Effectue un changement dans la vue *)
let replace_val j v (sheet : spreadsheet) (view : spreadview) dpd deg =
  sheet.(0).(j) <- Value v ;
  view.(0).(j) <- v ;
  (* On n'invalide que si la cellule n'est pas déjà invalidée *)
  if deg.(0).(j) = 0 then
    (* On invalide toutes les cellules qui dépendent de la valeur en 0:j *)
    invalidate_deps dpd deg (0,j)

(* Somme des degrés *)
let sum_deg (deg : status) =
  let n = Array.length deg in
  let m = Array.length deg.(0) in
  let sum = ref 0 in
  for i = 0 to n - 1 do
    for j = 0 to m - 1 do
      sum := !sum + deg.(i).(j)
    done
  done ;
  !sum

(* Nombre de cellules invalidées (deg <> 0) *)
let count_dirty (deg : status) =
  let n = Array.length deg in
  let m = Array.length deg.(0) in
  let sum = ref 0 in
  for i = 0 to n - 1 do
    for j = 0 to m - 1 do
      if deg.(i).(j) <> 0 then incr sum
    done
  done ;
  !sum
```

Question à développer pendant l'oral 5 *Quelle est la complexité en temps dans le pire cas de*

cette première passe d'invalidation en fonction des dimensions d'un tableur n et m ? Qu'observe-t-on empiriquement au regard des valeurs demandées ci-dessus ? On pourra séparer la phase d'initialisation dans l'analyse de complexité.

Le remplacement de la valeur d'une cellule revient à visiter tous les arcs du sous-graphe du graphe de dépendances engendré par la cellule modifiée et ses successeurs.
Dans le pire des cas, toutes les cellules de la colonne 1 dépendent de la même cellule invalidée. Ainsi, toutes les cellules contenant des formules se trouvent elles aussi invalidées. Cela donnerait une complexité en temps linéaire : $\mathcal{O}(n \times m)$.
Ceci dit, on peut observer empiriquement un nombre de cellules invalidées faible devant les dimensions du tableur. Cela est dû à la façon dont ils sont générés.
Si l'on ignore la phase d'initialisation des degrés (qui est en temps linéaire) qui peut être réalisée de concert avec l'évaluation initiale du tableur, on a un parcours qui semble économique en pratique.

4.3 Passe de réparation

La deuxième passe effectue la réparation de la vue en partant des cellules dépendant directement des cellules modifiées. Pour chaque cellule visitée, on décrémente le nombre de dépendances invalides, et si ce nombre atteint 0, alors on peut évaluer la cellule à nouveau sans risque (toutes ses dépendances ont été réparées).
En partant du tableur modifié en Figure 6, on commence par remettre le degré de $(0,2)$ à 0 avant de visiter les cellules $(1,2)$ et $(1,1)$ qui en dépendent. $(1,2)$ a pour degré 2, nombre que l'on décrémente à 1, signifiant qu'une de ses dépendances n'a toujours pas été réparée. On visite alors $(1,1)$ qui est de degré 1, on peut donc descendre ce degré à 0 et effectuer le calcul, ce qui donne 5. On visite maintenant les dépendances de $(1,1)$, c'est-à-dire $(1,2)$ à nouveau. On descend son degré à 0 et on calcule sa valeur : 0. Ceci donne ainsi la vue réparée en Figure 7.

	0	1	2	3	4
0	3	8	1	1	8
1	2	5	9	5	9
2	6	0	0	10	7

FIGURE 7 – Vue réparée

Nous allons maintenant effectuer la réparation des vues pour les quatre tableurs que nous avons modifiés ci-dessus. Votre algorithme devra commencer par propager les changements à partir des dépendances des dix cellules modifiées.

Question 7 *Calculez la somme modulo p des valeurs des cellules dans la vue associée au tableur* $\mathsf{T}(\mathsf{faux}, n, m, p)$ *après l'avoir réparée pour les valeurs de n, m et p suivantes.*

a) $n = 10$, $m = 10$, $p = 97$

b) $n = 10$, $m = 100$, $p = 547$

c) $n = 30$, $m = 1000$, $p = 1091$

d) $n = 1000$, $m = 2000$, $p = 5099$

```
(* Évaluation sans cycle *)
let eval_sheet p sheet : spreadview =
  let n = Array.length sheet in
  let m = Array.length sheet.(0) in
  let view = Array.make_matrix n m 0 in
  for i = 0 to n - 1 do
    for j = 0 to m - 1 do
      view.(i).(j) <- eval_cell p sheet view i j
    done
  done ;
  view

(* Contrôle d'une vue *)
let view_checksum p (view : spreadview) =
  let n = Array.length view in
  let m = Array.length view.(0) in
  let sum = ref 0 in
  for i = 0 to n - 1 do
    for j = 0 to m - 1 do
      sum := (!sum + view.(i).(j)) mod p
    done
  done ;
  !sum
```

```
(* Propage les changements sur la cellule (i,j) si possible *)
let rec propagate p sheet view (dpd : dpdgraph) (deg : status) (i,j) =
  deg.(i).(j) <- deg.(i).(j) - 1 ;
  if deg.(i).(j) = 0 then begin
    view.(i).(j) <- eval_cell p sheet view i j ;
    propagate_deps p sheet view dpd deg i j
  end

(* Propage les changements sur successeurs de (i,j) *)
and propagate_deps p sheet view dpd deg i j =
  List.iter (propagate p sheet view dpd deg) dpd.(i).(j)

(* On appelle la propagation sur les changements effectués *)
let test_propagate p sheet view dpd deg seed =
  let m = Array.length sheet.(0) in
  for k = 0 to 9 do
    let j = safe_un (seed + 2 * k) mod m in
    if deg.(0).(j) = 1 then begin
      deg.(0).(j) <- 0 ;
      propagate_deps p sheet view dpd deg 0 j
    end
  done
```

Question à développer pendant l'oral 6 *Commentez la complexité des deux passes en fonction des dimensions, mais aussi en fonction des données du graphe de dépendances.*

La complexité en temps de la passe d'invalidation a déjà été abordée. La passe de réparation va visiter autant de fois une cellule que son degré d'invalidation, nombre lui-même borné par le nombre d'arcs dans le graphe de dépendances comme énoncé précédemment. On opère de nouveau en temps linéaire : $\mathcal{O}(n \times m)$. Ceci dit, si le nombre d'arcs dans le graphe de dépendances est plus petit (comme on l'observe empiriquement), on gagne en efficacité.

Question à développer pendant l'oral 7 *À quel point l'efficacité de l'approche présentée ici est dépendante de la génération aléatoire des données ?*

Comme déjà évoqué, dans le pire cas, la cellule modifiée pourrait nécessiter de recalculer l'entièreté de la vue (mis à part le reste de la première colonne), ce qui en réalité coûterait marginalement plus cher (même si l'on reste dans le même ordre de complexité).
La génération ici permet d'éviter cette situation. En effet, en moyenne, en ignorant la première colonne, un tiers des cellules fait référence à deux autres, un autre tiers à une seule, et le dernier tiers à aucune. Ceci donne en moyenne une cellule référencée par cellule. On évite donc une explosion combinatoire que l'on aurait par exemple en passant ce nombre à deux.

5 Grandes feuilles de calcul

Dans cette partie, on considère des feuilles de calcul de très grande taille.

L'objectif de cette partie est de calculer la valeur d'un petit nombre de cellules, en l'occurrence les trois premières lignes de la dernière colonne. Par exemple pour un tableur de dimensions 10×27, cela correspond aux cellules $(9, 0)$, $(9, 1)$ et $(9, 2)$.

Question 8 *Calculez la somme modulo p des valeurs des trois premières cellules de la dernière colonne dans la vue associée au tableur $\mathsf{T}(\mathsf{faux}, n, m, p)$ pour les valeurs de n, m et p suivantes.*

- ***a)*** $n = 1000$, $m = 20\,000$, $p = 97$
- ***b)*** $n = 10\,000$, $m = 34\,000$, $p = 127$
- ***c)*** $n = 70\,000$, $m = 567\,000$, $p = 257$
- ***d)*** $n = 234\,000$, $m = 9\,170\,000$, $p = 307$

```ocaml
(* On ne peut pas construire le tableur cette fois *)
let get_cell (n,m,p) i j =
  if i = 0 then Value ((safe_un j) mod p)
  else gen_cell false n m p i j

(* Vue partielle creuse *)
type sparse_view = (cref, int) Hashtbl.t

let view_init size : sparse_view =
  Hashtbl.create size

let read_view (view : sparse_view) i j =
  Hashtbl.find_opt view (i,j)

let add_view (view : sparse_view) i j v =
  if not (Hashtbl.mem view (i,j)) then
    Hashtbl.add view (i,j) v

let rec sparse_eval p sheet view (i,j) =
  match read_view view i j with
  | Some v -> v
  | None ->
    begin match get_cell sheet i j with
    | Value v -> add_view view i j v ; v
    | Formula f ->
        let v = sparse_eval_formula p sheet view f in
        add_view view i j v ;
        v
    end

and sparse_eval_formula p sheet view =
  function
  | PLUS (c1,c2) ->
    (sparse_eval p sheet view c1 + sparse_eval p sheet view c2) mod p
  | TIMES (c1,c2) ->
    (sparse_eval p sheet view c1 * sparse_eval p sheet view c2) mod p
  | MINUS c ->
    p - sparse_eval p sheet view c
```

Question à développer pendant l'oral 8 *Décrivez l'algorithme ainsi que les structures de données utilisés.*

On construit la vue au fur et à mesure en maintenant une table de hachage (ou une table d'association, du moment que les accès sont suffisamment efficaces) qui à chaque référence de cellule associe sa valeur si elle est déjà connue. Quand on évalue une cellule, on regarde d'abord si sa valeur n'a pas déjà été calculée, si oui on renvoie cette valeur, sinon on procède récursivement.

6 Mesure d'impact

On s'intéresse pour finir au nombre de cellules qui se retrouveraient affectées par le changement d'une cellule donnée. Autrement dit, on veut calculer combien de cellules dépendent directement et indirectement d'une cellule donnée. On appellera ce nombre l'impact de la cellule en question.

Si on reprend l'exemple de la Figure 1 on observe que l'impact de la cellule $(0, 0)$ est de 3 car les cellules $(2, 0)$, $(3, 2)$ et $(4, 1)$ en dépendent.

Plus précisément, la question que l'on se pose est de savoir combien de cellules ont un impact strictement supérieur au nombre de colonnes n.

Question 9 *Calculez le nombre de cellules à impact strictement supérieur au nombre de colonnes n dans les tableurs suivants.*

- ***a)*** $\mathsf{T}(\mathsf{faux}, 10, 100, 97)$
- ***b)*** $\mathsf{T}(\mathsf{faux}, 200, 300, 151)$
- ***c)*** $\mathsf{T}(\mathsf{faux}, 1000, 1000, 251)$
- ***d)*** $\mathsf{T}(\mathsf{faux}, 1000, 3000, 373)$
- ***e)*** $\mathsf{T}(\mathsf{faux}, 1000, 10\,000, 7013)$
- ***f)*** $\mathsf{T}(\mathsf{faux}, 500, 30\,000, 3931)$

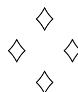
```
type succs =
| Enough
| Exactly of int * (int * int) list

let insert_succs target x u =
  match u with
  | Enough -> Enough
  | Exactly(_,[]) -> Exactly(1,[x])
  | Exactly(n,y :: l) when x = y -> u
  | Exactly(n,l) -> if n+1 > target then Enough else Exactly(n+1, x :: l)

let rec merge_len target n l1 m l2 : succs =
  match l1, l2 with
  | x :: l1, y :: l2 ->
    if x < y then insert_succs target x (merge_len target (n-1) l1 m (y :: l2))
    else insert_succs target y (merge_len target n (x :: l1) (m-1) l2)
  | [], l | l, [] -> Exactly(n+m, l) (* n ou m = 0 *)

let merge_succs target u v =
  match u, v with
  | Enough, _ | _, Enough -> Enough
  | Exactly(n,l1), Exactly(m,l2) -> merge_len target n l1 m l2

let q9 n m p =
  let sheet = gen_spreadsheet false n m p in
  let succ = Array.make_matrix n m (Exactly (0,[])) in
  let add_succ (i,j) (k,l) =
    let m = merge_succs n succ.(i).(j) succ.(k).(l) in
    succ.(k).(l) <- insert_succs n (i,j) m
  in
  for i = n-1 downto 0 do
    for j = m-1 downto 0 do
      begin match sheet.(i).(j) with
      | Value i -> ()
      | Formula (PLUS (c1,c2))
      | Formula (TIMES (c1,c2)) ->
        add_succ (i,j) c1 ;
        add_succ (i,j) c2
      | Formula (MINUS c) ->
        add_succ (i,j) c
      end
    done
  done ;
  let sum = ref 0 in
  for i = 0 to n-1 do
    for j = 0 to m-1 do
      if succ.(i).(j) = Enough then incr sum
    done
  done ;
  !sum
```

Question à développer pendant l'oral 9 *Expliquez l'algorithme que vous avez utilisé.*

Une première version naïve consiste à calculer pour chaque cellule, la liste des cellules qui en dépendent directement ou indirectement, en prenant soin de retirer les doublons. Il suffit ensuite de comparer la longueur de ces listes à n et de compter le nombre de cellules pour lequel elle est strictement plus grande.
Pour rendre l'algorithme plus efficace, nous faisons une série d'optimisations :

- Pour commencer, on parcours le tableur "à l'envers", en commençant par la dernière ligne de la dernière colonne. De cette manière on les visite dans un ordre inverse à l'ordre topologique, ce qui assure que toutes les cellules qui dépendent de la cellule courante ont été visitées avant.
- Le point précédent s'assure que nous n'avons pas besoin du graphe de dépendances. À la place, pour chaque cellule, on va aller ajouter la liste de ses successeurs aux cellules dont elle dépend directement.
- Pour s'assurer de ne pas compter les doublons, on va considérer que les listes de successeurs sont triées. Pour ça, on utilise des fonctions d'insertions et de fusion qui préservent le tri plutôt que l'ajout en tête de liste et la concaténation.
- Comme on rend visite à chaque cellule dans un ordre lexicographique décroissant, l'insertion peut en réalité se faire en temps constant si la liste est triée dans le même ordre.
- Enfin, une fois que la liste a atteint la masse critique de $n+1$, il n'est plus nécessaire de la conserver. On ne va donc pas stocker forcément des listes mais en réalité soit l'information que l'on a atteint la masse critique, soit une liste accompagnée de sa longueur pour éviter de la recalculer sans cesse.

D'autres pistes sont bien évidemment possibles.



Fiche réponse type : Calculs et tableurs

$\widetilde{u_0}$: 237

Question 1

a) 924

b) 915

c) 743

d) 944

Question 2

a) 4696

b) 71

c) 216

d) 432

Question 3

a) 4367

b) 4010

c) 9312

d) 4185

Question 4

a) 54

b) 589

c) 18 380

d) 58 704

Question 5

a) 2

b) 481

c) 447

d) Cycle

Question 6

a) (24, 23)

b) (85, 85)

c) (412, 408)

d) (94, 94)

Question 7

a) 11

b) 121

c) 537

d) 2645

Question 8

a) 4

b) 112

c) 27

d) 264

Question 9

a) 146

b) 3093

c) 20 226

d) 71 479

e) 227 260

f) 494 959