ÉPREUVE PRATIQUE D'ALGORITHMIQUE ET DE PROGRAMMATION DU CONCOURS COMMUN DES ÉCOLES NORMALES SUPÉRIEURES — RAPPORT DE JURY 2018

École concernées : Cachan, Lyon, Paris, Rennes
Jury : Pierre-Évariste Dagand, Nathanaël Fijalkow, Vincent Gripon, Ocan Sankur

Organisation de l'épreuve

L'objectif de cette épreuve est la capacité de mettre en œuvre une chaîne complète de résolution d'un problème informatique, à savoir la construction d'algorithmes, le choix de structures de données, leurs implémentations, et l'élaboration d'arguments mathématiques pour justifier ces décisions. Le déroulement de l'épreuve est le suivant : un travail sur machine d'une durée de 3h30, immédiatement suivi d'une présentation orale pendant 25 minutes.

Un sujet contient typiquement une dizaine de questions, de deux types : écrites et orales. Tous les sujets commencent par la génération pseudo-aléatoire d'entrées pour le problème étudié. Nous invitons fortement les candidats à se familiariser 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. En particulier, les suites pseudo-aléatoires dépendent d'un u_0 qui est donné individuellement à chaque candidat au début de l'épreuve.

Les questions écrites attendent des réponses purement numériques. Chaque question requiert l'implémentation d'un algorithme et son utilisation sur des entrées pseudo-aléatoires générées au début du sujet. Une question est typiquement divisée en sous-questions pour des entrées de plus en plus grandes, ce qui permet de tester l'efficacité de l'algorithme mis en œuvre. À la fin de la partie pratique, les candidats remettent à l'examinateur une fiche réponse contenant les réponses aux questions écrites (purement numériques).

Une aide précieuse est donnée aux candidats sous la forme d'une fiche réponse type pour $\widetilde{u_0}$. Cette fiche permet de vérifier l'exactitude des réponses pour une graine différente du u_0 du candidat. 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. Les examinateurs ont encore eu quelques (rares, heureusement) cas de candidats traitant le sujet avec un générateur faux et donc sans possibilité de diagnostiquer efficacement leurs erreurs.

Les questions orales sont de nature plus théorique et sont destinées à être présentées pendant l'oral. La présentation orale vise à évaluer la bonne compréhension du sujet et le recul des candidats. Les examinateurs s'efforcent daborder 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 les solutions au tableau.

Juste avant la distribution des sujets, les candidats disposent d'une période de 10 minutes pour se familiariser avec l'environnement informatique et poser des questions aux surveillants s'ils rencontrent des difficultés d'ordre pratique.

La partie pratique de l'épreuve représentait cette année de l'ordre de 60% de la note finale, le coefficient exact variant suivant le sujet. On observe dans l'ensemble, mais pas systématiquement, une bonne corrélation entre les résultats obtenus aux deux parties.

Remarques générales

Dans certains cas, les examinateurs ont inspecté le code des candidats afin de lever certaines ambiguïtés lors de leur présentation de leurs algorithmes. Cela n'était possible uniquement pour les candidats qui avaient soigné leur code dans lequel les calculs produisant les réponses aux questions étaient facilement identifiables et exécutables. Nous conseillons ainsi aux candidats de soigner la lisibilité de leur code. Les candidats peuvent s'inspirer de deux propositions de corrigés fournis en annexe de ce rapport.

La durée de l'oral étant courte relativement au nombre de questions pouvant être traitées, nous conseillons aux candidats de préparer une réponse précise mais intuitive plutôt que de se perdre dans une preuve laborieuse au tableau. Si l'examinateur n'est pas convaincu par un argument simple, il sera toujours possible de le convaincre par une preuve détaillée sans que cela impacte la note finale. Inversement, si l'examinateur est convaincu par un raisonnement intuitif, le candidat dispose alors de plus de temps pour aborder des questions globalement peu traitées et donc susceptibles de rapporter beaucoup de points. Par exemple, nombre de candidats se lancent dans d'interminables preuves par induction alors qu'il existe parfois une explication intuitive immédiate : nous encourageons les candidats à favoriser la seconde. La capacité à exposer un argument formel pour répondre à une question est évalué dans le cadre de l'épreuve d'informatique fondamentale, alors que l'objet de l'oral du TP informatique est de s'assurer que les candidats font le lien entre la résolution d'un problème informatique dans un cadre formel inédit, l'examinateur saura donc apprécier le recul que démontre un argument simple et intuitif par rapport à une suite d'arguments formels désincarnés.

De même, certaines questions orales demandent aux candidats de présenter leur algorithme et d'analyser leur complexité. Nous encourageons vivement les candidats à présenter leurs algorithmes de façon claire et concise. Contrairement à ce que nous avons fréquemment pu constater, il ne s'agit pas nécessairement de recopier un programme en pseudo-Python ou pseudo-Caml au tableau : il faut s'efforcer de présenter (uniquement) les étapes clés de l'algorithme, en langage naturel si nécessaire, afin de supporter efficacement l'analyse de complexité par la suite. En particulier, il est essentiel d'identifier clairement la structure itérative ou récursive d'un algorithme. Trop souvent, des candidats se sont trompés entre une complexité en $O(m+n)$ et une complexité en $O(m \times n)$ à cause d'une présentation de l'algorithme trop confuse. On visera donc à être à la fois concis sans pour autant sacrifier la précision de la présentation. Enfin, ce type de question ne doit pas empêcher un étudiant de proposer un algorithme simple (accompagné d'une analyse de complexité correcte) même si l'implémentation de l'algorithme en question n'a pas été achevée durant la partie pratique de l'épreuve. Si l'algorithme proposé est en réalité trop

naïf pour traiter les instances proposées dans le sujet, l'examinateur saura apprécier un regard critique sur l'algorithme qui exploiterait l'analyse de complexité.

Nous terminons par un exemple : la présentation de l'algorithme de parcours de graphe. Voici les quatre phrases que l'on attend 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 mise 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 $O(n + m)$, où n est le nombre de sommets et m le nombre d'arêtes.

Sujet 1 : Analyses de programmes

Ce sujet s'intéressait à l'analyse de programmes écrits dans un langage pseudo-assembleur afin d'identifier et d'éliminer du "code mort", c'est-à-dire des instructions n'ayant aucun effet sur le résultat final du programme.

À l'écrit, la première partie du sujet (questions 1 à 4) était un exercice de programmation où il s'agissait pour l'essentiel d'implémenter correctement la spécification fournie. Plus de 80% des candidats ont traités correctement ces questions, les questions 1 à 3 étant traitées sans erreur par tous les candidats.

La seconde partie du sujet (question 5) demandait un effort d'abstraction consistant à voir un programme comme un graphe. Cette question a été très bien traitée, avec plus de 80% de réussite.

Surprenamment, la troisième partie du sujet a posé d'immenses difficultés. La question 6, qui consistait à implémenter un algorithme de calcul de point fixe donné dans l'énoncé, n'a été correctement traité que par 17% des candidats. L'examen des programmes des candidats ayant réussi la question 5 et échoué à la question 6 semble indiquer des difficultés, particulièrement en Python, à maintenir deux copies *distinctes* des ensembles de variables vivantes afin de détecter la convergence. Ceci est d'autant plus regrettable qu'une solution détaillée pour la graine $\tilde{u_0}$ était fournie dans l'énoncé. Nous exhortons les candidats à tester leurs programmes avec les exemples donnés dans le sujet, quitte à exécuter l'algorithme "à la main" et à écrire leurs propres tests lorsque leur solution diverge de celle donnée dans la fiche réponse.

La question 7 exigeait un effort d'abstraction, guidé par la question orale 5. Sans surprise eu égard au traitement mitigé de la question orale 5, la question 7 a été traitée par seulement 1 candidat.

La quatrième partie (question 8) consistait à identifier la première instruction morte d'un programme pseudo-aléatoirement généré. Afin de traiter rapidement cette question, il fallait remarquer (sans indication du sujet) que le résultat de l'analyse de vivacité permet d'identifier simplement (et efficacement) de telles instructions. Algorithmiquement, cette question ne posait pas de difficulté et a été réussie par 3 candidats.

Aucun candidat n'a abordé la dernière partie, qui consistait à déterminer comment mettre à jour de façon incrémentale le résultat de l'analyse de vivacité après élimination d'une instruction. En effet, la taille des instances considérées rendait impossible la stratégie naïve consistant à modifier le programme puis relancer l'analyse de vivacité de façon répétée.

À l'oral, les questions orales 1 et 2 ont été globalement bien traitées par l'ensemble des candidats (plus de 80% de réussite). On regrettera cependant que certains candidats confondent encore matrice d'adjacence et liste d'adjacence, décrivant l'un sous le nom de l'autre : il s'agit de questions de cours, pour lesquelles nous attendons 100% de réussite.

La question orale 3 a donné lieu à des réactions mitigées. La preuve de terminaison et la justification du fait que l'exemple donné dans l'énoncé est bien la plus petite solution n'ont pas posé de problème (plus de 70% de réussite). Cela a agréablement surpris les examinateurs. Cependant, la preuve que la solution calculée par l'algorithme est minimale fut plus laborieuse, avec seulement 35% des candidats ayant fait un sans faute. Il suffisait pourtant de considérer n'importe quelle solution du système d'équation et de montrer que l'ensemble calculé par l'algorithme restait inclus dans cette solution.

La question orale 4 a été mieux traitée, avec quasiment 70% de réussite. Notons qu'il y avait deux réponses acceptables. On pouvait borner le nombre de registres utilisés par un programme de taille p par $O(p)$: les opérations sur les ensembles de variables ont alors une complexité en $O(p)$. Alternativement, on pouvait remarquer qu'un programme ne peut utiliser que 2048 registres : les opérations ensemblistes sont alors à coût constant.

Les questions orales 5 et 6 demandaient du recul sur la notion de vivacité afin d'établir, d'une part, un lien avec l'interférence et, d'autre part, un lien avec le code mort. Environ 10% des candidats ont réussi à traiter ces questions sans hésitation. Malheureusement, trop de candidats n'ont pas su exploiter les parties précédentes et ont proposés divers parcours (faux) sur le graphe de flot de contrôle, oubliant la présence de boucles et donc la nécessité de calculer un point fixe.

Dans la dernière partie, seule la question orale 7 a été traitée correctement par 3 candidats. Les examinateurs ont récompensé de bonnes propositions, même si celles-ci n'avaient pas été implémentées.

Le candidat ayant obtenu la meilleure note a traité toutes les questions jusqu'à la partie 4 (élimination de code mort) incluse et a traité correctement toutes les questions orales jusqu'à la partie 5.1 (élimination d'écritures) incluse.

Sujet 2 : (ppv)

Le sujet s'intéressait à la recherche du vecteur, dans un ensemble donné, le plus proche au sens de la distance Euclidienne d'un vecteur requête. Le sujet a été presque entièrement terminé par un candidat, alors que les questions 5 et 6 à l'écrit ont été des paliers pour la plupart des autres.

Tous les candidats ont répondu correctement à la question 1 qui testait le générateur pseudo-aléatoire. La question 2 a également été correctement traitée pour la quasi-totalité des candidats. Cette question avait pour principal objectif de s'assurer que les candidats avaient bien pris note du fait qu'en cas d'égalité de distances entre le vecteur requête et plusieurs vecteurs dans l'ensemble de recherche, seul celui d'indice le plus faible devait être retourné.

Les candidats ayant partiellement échoué à la question 2 ont naturellement également échoué à la question 3, qui s'appuyait essentiellement sur le code produit pour répondre à la question 2. Aussi, les candidats n'ayant pas correctement tabulé les valeurs du générateur pseudo-aléatoire ont rencontré les premières difficultés de temps de calcul. Au final, les moyennes pour les questions 1 à 3 sont respectivement 100%, 94% et 88%.

Beaucoup des candidats ont utilisé la solution de la question 3 pour répondre à la question 4, ce qui était possible mais requiérait un temps de calcul important. On pouvait plutôt déterminer pour chaque vecteur $\mathbf{y}$ requête l'intervalle des vecteurs $\mathbf{z}$ pour lesquels $\mathbf{y}$ augmentait la représentativité de $\mathbf{z}$. Cette question a eu pour moyenne 81%.

La question 5 a été un premier palier pour les candidats, avec une moyenne de 51% des points obtenus pour l'ensemble des candidats. Il s'agissait d'une question demandant de tester la connexité de graphes générés à partir de vecteurs. Il fallait plus précisément trouver le plus petit paramètre k pour lequel le graphe devenait connexe. La plupart des candidats avaient bien identifié que les graphes étaient croissants (au sens de l'inclusion des arêtes) avec k, mais très peu ont eu l'idée d'utiliser une recherche dichotomique. La plupart des candidats ont en revanche bien pensé à mettre en place un parcours de graphe pour vérifier la connexité.

La question 6 n'a rapporté que 20% des points en moyenne. Elle n'apportait pourtant pas de difficulté particulière. Il s'agissait encore d'effectuer une recherche dichotomique pour trouver un paramètre k limite.

La question 7 apportait une difficulté supplémentaire. Il s'agissait de recherche la plus grande clique dans un graphe, ce qui est un problème difficile bien connu en algorithmique des graphes. Le très faible nombre de candidats ayant résolu cette question ont pensé à utiliser une recherche par extension, en partant de l'ensemble vide est en testant toutes les façons d'aggrandir l'ensemble.

Les questions 8 et 9 avaient pour objectif de faire identifier aux candidats des cas particuliers pour lesquels la recherche du plus proche vecteur pouvait se faire par indexation de l'espace de recherche. En effet dans ces deux cas, l'espace entier était suffisamment petit pour tenir en mémoire. Les candidats ayant répondu au moins partiellement à une de ces deux questions se comptent sur les doigts d'une main. Ces trois dernières questions ont eu pour moyennes 14%, 5% et 5%.

À l'oral, des difficultés non anticipées ont provoqué un effondrement des résultats sur la question 2, ce qui dans de nombreux cas a monopolisé un temps important. Si la question 1 demandait d'estimer une complexité et a été traitée avec succès, la question 2 portait également sur la complexité mais demandait cette fois de montrer qu'il existait un minorant et n'a pas – ou a très mal – été traitée par la quasi-totalité des candidats.

Cette question a été révélatrice du fait que trop de candidats ont une incompréhension fondamentale de la notion de complexité, et que s'ils savent estimer la complexité d'un algorithme en multipliant les cardinaux des boucles dans un algorithme, ils peinent à manipuler la notion dans un contexte légèrement différent. Pour répondre à cette question, il était attendu que les candidats montrent qu'un algorithme n'explorant qu'une partie des coordonnées des vecteurs requêtes pouvait être trompé. La question 1 a rapporté 98% des points en moyenne et la question 2 30%.

La question orale 3 n'a pas posé de difficultés (100% de réussite), et n'était là que pour aider à la résolution de la question 4 écrite. La question orale 4 a par contre été parfois mal traitée au niveau de l'analyse de complexité. Certains candidats ont en effet indiqué que la complexité d'un parcours en profondeur était en $\mathcal{O}(|V||E|)$ [1]. Elle a rapporté 58% des points en moyenne.

La question orale 5 demandait de trouver des cas extrêmes dans un cas simple. Le premier exemple a été trouvé par un très grand nombre de candidats, alors que le second cas n'a été trouvé que par une plus faible proportion d'entre eux (respectivement 86% et 57%).

Il était essentiellement attendu à la question 6 que les candidats nous disent qu'un sous-ensemble de sommets d'une clique forme également une clique. Certains candidats ont tenté des preuves par récurrence sur les étapes de l'algorithme, souvent sans succès. La moyenne obtenue est de 39%.

Pour les dernières questions orales, un nombre relativement important de candidats avaient de bonnes idées sur comment résoudre les questions écrites, mais n'avaient pas réussi ou avaient manqué de temps pour produire un code fonctionnel. Elles ont rapporté 23% et 12% des points respectivement.

Un exemple de solution pour cet exercice est disponible en annexe à ce document.

Sujet 3 : (Election de leader)

Le sujet traitait deux algorithmes d'élection de leader utilisés dans les systèmes distribués pour distinguer un processus dans le réseau.

0.1. **Première partie.** Les deux premières questions concernaient la construction de la suite pseudo-aléatoire et celle des graphes définissant les systèmes distribués, et ne présentaient pas de difficulté particulière. A la question 3, il fallait simplement implémenter l'algorithme décrit dans le sujet. Il s'agissait d'un petit programme qui s'exécute dans la machine sélectionnée à chaque étape. Il fallait faire attention à bien utiliser une file pour stocker les messages reçus par chaque machine, et penser à inclure l'identifiant de l'emetteur d'un message dans ces files. Cette question a été correctement traitée par 2/3 des candidats. Cet algorithme effectue un parcours dans les graphes des machines, et construit un arbre qui est représenté du bas en haut : chaque machine dispose de la variable `parent` qui est l'identifiant du parent dans cet arbre en construction. La question 4 demandait un calcul sur cet arbre ainsi calculé. La seule difficulté était d'obtenir d'abord une représentation habituelle (de haut en bas) de l'arbre. 2/3 des candidats ont réussi cette question.

La question d'oral 1 est une question de cours sur le parcours en largeur. Il fallait décrire cet algorithme et donner sa complexité, et montrer comment obtenir une implémentation avec cette complexité. 2/3 des candidats avaient la bonne réponse. On rappelle que l'algorithme est en temps linéaire en nombre d'arêtes et non pas en nombre de sommets. Beaucoup de candidats sont tombés dans ce piège, et n'étaient pas capables de faire une analyse de complexité correcte d'un algorithme simple qui est pourtant au programme.

La question d'oral 2 demandait à borner la variable `distance` et en déduire une borne sur le temps de terminaison. La moitié des candidats ont réussi cette question en remarquant que chaque valeur d'une variable `distance` correspondait à un chemin simple, et qu'un message ne pouvait pas suivre un cycle sur le graphe.

1. Dans l'absolu ce résultat n'est pas faux, mais loin du $\mathcal{O}(|V| + |E|)$ attendu.

A peu près 2/3 des candidats ont compris cette idée et ont réussi à borner la variable `distance`. Une grande majorité a réussi à en déduire une borne sur la terminaison. Comme d'habitude l'analyse de complexité s'avère toujours compliquée et les candidats sont rarement rigoureux : seulement un tiers des candidats ont fait une analyse de complexité correcte.

0.2. **Deuxième partie.** Cette partie traitait de l'algorithme de Herman, qui est un algorithme d'élection de leader pour les réseaux disposés en anneau. Les questions 5 et 6 consistent à implémenter l'algorithme décrit dans le sujet. Une majorité des candidats ont bien répondu à ces questions : 85 % pour Q5, et 70 % pour Q6.

Le reste du sujet comportait des questions plus difficiles. Pour répondre à la question 7, il fallait remarquer la taille relativement faible du nombre de jetons dans les instances données. Une recherche exhaustive par parcours en largeur était suffisante pour trouver l'ordonnanceur minimal. Moins d'un tiers des candidats ont bien répondu à cette question.

Dans la question 8, on revenait à la question 6 mais pour des instances de très grande taille. Une implémentation immédiate de l'algorithme du sujet ne permettait pas de répondre à cette question. Il fallait remarquer plusieurs astuces. Par exemple on n'a pas besoin de stocker toute la configuration puisque la position des jetons suffisent à calculer l'évolution du réseau. Seulement 6 candidats sur 39 ont répondu juste ou partiellement juste à cette question.

La question 9 était très difficile et n'a été traitée par aucun candidat.

La question d'oral 3 permettait de distinguer le coût de chaque étape de l'exécution de l'algorithme. On obtenait souvent une complexité linéaire puisqu'il fallait trouver le premier jeton, et échanger eventuellement le bit en temps constant. On pouvait remarquer que sous l'ordonnanceur minim, on faisait un nombre linéaire d'opérations pour trouver le même jeton un grand nombre de fois, ce qui pouvait inciter à garder en mémoire plutôt la position des jetons et non pas le tableau entier. Une grande majorité des candidats ont bien répondu à cette question.

La question d'oral 4 demandait à démontrer que chaque étape de l'algorithme faisait décroître la configuration pour un ordre bien fondé. Plus de 3/4 des candidats ont bien répondu à cette question. On en déduisait que l'algorithme termine avec l'ordonnanceur minim. On pouvait aussi en déduire une borne précise du temps de terminaison et s'en servir pour calculer le temps de terminaison sans exécuter l'algorithme (voir la question 8).

La question d'oral 5 était facile puisqu'il suffisait de remarquer que l'algorithm ne termine pas avec un ordonnanceur constant. 3/4 des candidats ont bien répondu à cette question.

La question d'oral 6 demandait l'algorithme et la complexité de la solution de la question 7. Seulement 1/3 des candidats ont bien répondu à cette question. Il fallait remarquer qu'une recherche exhaustive pouvait être exponentielle mais en fonction de nombre de jetons plutôt qu'en fonction de la taille du réseau. Les candidats qui ont inspecté les instances générées dans la question 7 ont pu voir qu'il y avait toujours un très petit nombre de jetons.

A la question d'oral 7, il s'agissait de commenter la solution à la question 8. 1/5 des candidats ont répondu correctement.

La dernière question d'oral portait sur la solution à la question 9. Seulement trois candidats ont trouvé la bonne solution (complètement ou partiellement) qui était un algorithme de programmation dynamique.

Sujet 4 : Sous-suites

Ce sujet s'intéressait à l'extraction de sous-suites particulières d'une ou deux suites de nombres. Étant donnée une suite de nombres, il s'agissait d'extraire une plus longue sous-suite croissante (partie 1) et une plus lourde sous-suite (partie 4). Étant données deux suites de nombres, il s'agissait d'extraire une plus longue sous-suite commune aux deux (partie 2) et une plus lourde sous-suite commune (partie 4).

La première partie du sujet (questions 1 et 2) consistait à implémenter correctement des générateurs pseudo-aléatoires à partir d'une spécification fournie. La seconde partie du sujet (question 3) visait à déterminer la longueur d'une plus longue sous-suite croissante d'une suite donnée, un algorithme étant fortement suggéré par le sujet. Ces trois questions ont été correctement traitées par l'ensemble des candidats. La question orale 1 permettait d'étayer la relation entre l'algorithme suggéré et la question 3 : celle-ci a été correctement traitée par l'ensemble des candidats. La question orale 2 visait à vérifier que les candidats avaient réellement compris l'algorithme qu'ils avaient implémenté : de façon révélatrice, on constate que seul 40% des candidats avaient atteint ce niveau de compréhension. Cela ne prêtait pas à conséquence pour la partie 3 mais explique les mauvais résultats dans la partie 4 où il s'agissait de généraliser cet algorithme suite à l'introduction d'une fonction de poids. La question orale 3 demandait essentiellement une analyse de complexité sur un algorithme donné : elle fut bien traitée par 65% des candidats.

La deuxième partie portait sur l'extraction d'une plus longue sous-suite commune à deux suites. Le sujet prenait un tour expérimental avec la question 4, où l'on demandait de calculer un nombre d'indices en correspondance pour les deux suites considérées. D'un point de vue algorithmique, la question 4 n'a pas posé de problème (tous les candidats l'ont correctement traitée). Celle-ci servait de support à la question orale 4, qui demandait de comparer un scénario de pire cas (lorsque l'on considère des paires de suites quelconques) et les suites traitées dans le sujet, pour lesquelles la seconde suite est obtenue à partir de la première par une opération de mutation pseudo-aléatoire. Comme dans le sujet "Marches à petits pas dans le quart de plan" de l'année précédente, il s'agissait ici de formuler une observation à partir de résultats de simulation. Même si 80% des candidats ont fourni une réponse correcte, nous les encourageons à faire des propositions claires, avec conviction. Dans ce cas, nous attendions une estimation asymptotique tandis que certains candidats ont eu des difficultés à aller au-delà de "c'est beaucoup plus petit."

Le sujet introduisait ensuite la notion d'indice terminal du plus court préfixe d'une première suite ayant une sous-suite commune d'une longueur donnée avec le préfixe d'une seconde suite, à la fois sous la forme d'une spécification abstraite et d'une caractérisation récursive (notée $\mathrm{sp}(-,=)$). Il était admis que ces deux formulations étaient équivalentes. La question 5 visait à s'assurer que les candidats avaient réussi à implémenter cette fonction. Le résultat a été surprenamment nuancé : plus de 70% des candidats ont réussi à traiter la petite instance 5.a) tandis que seul 35% ont réussi à traiter l'instance 5.b) et moins de 5% a pu traiter 5.c). Pour traiter 5.b), il fallait remarquer que le calcul de la fonction récursive est aisément mémoïsable. Pour traiter 5.c), il fallait également remarquer qu'un tableau de taille $l+1$ suffisait pour calculer le résultat, ce qui évitait d'avoir à allouer une matrice de taille $(l+1)^2$. On pouvait également tirer parti de la question orale 6 (réussie par 80% des candidats)

pour effectuer des recherches dichotomiques dans $\text{corresp}(\sigma_1, i-1, \sigma_2)$ plutôt que linéaires dans $\text{sp}(i-1, -)$ pour déterminer l'indice j.

On demandait enfin d'obtenir la longueur d'une plus longue sous-suite commune à la question 6 pour des instances de tailles équivalentes puis supérieures à celles employées en 5.b) et 5.c). Plusieurs solutions étaient envisageables. La question orale 5 (réussie par plus de 80% des candidats) indiquait que ce problème est réductible au calcul de la longueur d'une plus longue sous-suite croissante, traité en seconde partie. Cela permettait de résoudre l'instance 6.a) de façon simple. Plus efficacement, la question 5 donnait directement la solution : il suffisait de déterminer le plus grand k tel que $\text{sp}(l, k) \neq \infty$. Seulement 2 candidats ont abordé cette question, en utilisant le résultat de la question 5. La question orale 7 a permis de recueillir les idées des candidats n'ayant pas nécessairement réussi à implémenter cette question : un peu moins de la moitié des candidats ont été en mesure de faire une proposition satisfaisante et une analyse de complexité correcte.

Il est compréhensible que des candidats aient implémentés $\text{sp}(-, =)$ de façon purement récursive (sans mémoïsation). En particulier, cela permettait de traiter 5.a) (où $l = 15$) et offrait la possibilité de tester des implémentations plus efficaces. Cependant, la question orale 7 a révélé que la majorité de ces candidats n'avaient pas conscience de l'inefficacité de leur implémentation, celle-ci étant inopérante au vu de la taille des instances considérées.

Dans la quatrième partie, on introduisait une notion de poids des éléments des suites considérées, le poids d'un élément étant dépendant de sa valeur et position dans la suite. La question 7 visait à calculer le poids d'une sous-suite de poids maximal. Plusieurs solutions étaient envisageables et discutées à la question orale 8. Six candidats ont astucieusement exploité la forme particulière de la fonction de poids pour réduire le problème du poids maximal à celui de la longueur maximale. Une autre solution consistait à généraliser l'algorithme glouton donné dans la seconde partie en manipulant des paires "(symbole terminal, poids de la sous-suite)", toujours de façon gloutonne. Aucun candidat n'a proposé spontanément cette solution mais plusieurs y sont parvenu lorsqu'il leur a été proposé de partir d'un exemple. La question 9 venait en écho à la question 7, où l'on demandait la longueur d'une sous-liste croissante de poids maximale et de longueur minimale. Un seul candidat a traité cette question. Une solution consistait à généraliser à nouveau l'algorithme glouton afin de manipuler des triplets "(symbole terminal, poids de la sous-suite, longueur de la sous-suite)" selon un ordre lexicographique sur le poids et l'inverse de la longueur.

La question 8 portait sur le calcul du poids d'une sous-suite commune de poids maximal tandis que la question 10 demandait la longueur d'une sous-suite commune de poids maximal et de longueur minimale. Il s'agissait ici de répéter la démarche de la partie 3, où l'extraction de sous-suite commune a été réduite à l'extraction d'une sous-suite des indices en correspondance. Un candidat a correctement traité la question 8 et partiellement traité la question 10. À la question orale 9, 4 candidats ont fait des propositions correctes pour résoudre la question 8, exploitant à nouveau et de façon inattendue la définition de la fonction de poids pour opérer une réduction de "plus lourde sous-suite commune" à "plus longue sous-suite commune". La question orale 10 n'a été abordée par aucun candidat.

Le candidat ayant obtenu la meilleure note a traité toutes les questions écrites sauf 5.c), 6.c), 9, 10.b) et 10.c) puis a traité de façon parfaite toutes les questions orales sauf la dernière.

Un exemple de solution pour cet exercice est disponible en annexe à ce document.

Annexe: exemple de solution pour l'exercice "Plus Proche Voisin"

6 septembre 2018

Cet exercice a été présenté à l'épreuve pratique d'algorithmique du concours des quatre Écoles Normales Supérieures le lundi 25 juin 2018. Il s'agit d'un exercice traitant de la recherche de plus proches voisins de vecteurs requêtes dans une collection de vecteurs références.

Nous avons observé une très grande variance dans les résultats des candidats, les moins avancés n'ayant pas réussi à passer la question 2 et le meilleur ayant presque tout validé sauf un cas de la question 7. Similairement, à l'oral le moins bon candidat n'a validé que les questions 1 et 3 alors que le meilleur a validé toutes les questions.

Ce corrigé n'est pas une correction parfaite des questions posées, mais a été rédigé avec l'intention de présenter des solutions utilisant peu de subtilités du langage python, et étant donc à la portée de tout étudiant de classes préparatoires. L'usage de la bibliothèque numpy en python était possible mais pas requise, et le corrigé n'en fait donc pas usage.

La plupart des candidats ont choisi de proposer une solution avec le langage OCaml (ou Caml-light) ou le langage Python.

1 Préliminaires

On rappelle que pour deux entiers naturels a et b, $a \mod b$ désigne le reste de la division entière de a par b.
Étant donné u_0, on définit par récurrence

$$\forall t \in \mathbb{N}, u_{t+1} = 19\,999\,999 u_t \mod 19\,999\,981 .$$

À partir de trois entiers positifs d, n, et m, on en déduit n vecteurs $(\mathbf{x}_i)_{0 \leq i < n}$ de dimension d, et m vecteurs $(\mathbf{y}_j)_{0 \leq j < m}$ de dimension d :

$$\forall 0 \leq k < d, \begin{cases} \mathbf{x}_i[k] = u_{id+k} \mod 1\,000, \\ \mathbf{y}_j[k] = u_{(n+j)d+k} \mod 1\,000, \end{cases}$$

où $\mathbf{x}_i[k]$ (respectivement $\mathbf{y}_j[k]$) désigne la k-ième coordonnée de $\mathbf{x}_i$ (respectivement $\mathbf{y}_j$). On notera dorénavant $\mathcal{X} = [\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}]$ la liste des vecteurs références et $\mathcal{Y} = \{\mathbf{y}_0, \ldots, \mathbf{y}_{m-1}\}$ l'ensemble des vecteurs requêtes. Par ailleurs, on notera $\mathbf{z}$ un vecteur quelconque dans la suite de l'énoncé.

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 votre (notée $\tilde{u}_0$). Il vous est conseillé de tester vos algorithmes avec $\tilde{u}_0$.

Question 1. *Indiquer les valeurs de* $\mathbf{x}_{33}[13]$ *et de* $\mathbf{y}_{123}[43]$ *dans les cas suivants : a) Pour* $n = 1\,000$, $m = 10\,000$ *et* $d = 128$, *b) Pour* $n = 1\,000$, $m = 10\,000$ *et* $d = 200$, *c) Pour* $n = 100$, $m = 10\,000$ *et* $d = 128$.

Comme dans beaucoup d'exercices, il est fortement conseillé aux candidats de tabuler les valeurs du générateur pseudo-aléatoire. Ici nous utilisons un tableau en Python. Compte tenu des questions de l'exercice, nous identifions que $40\,000\,000$ est une valeur suffisante.

```python
# Random seed
u0 = 42

# We index the first values of u
u = [u0]
for i in range(40000000):
    u.append((19999999 * u[-1]) % 19999981)

# Function to generate sets X and Y given d, n and m
def gen_sets(d, n, m):
    X = [[0 for k in range(d)] for i in range(n)]
    Y = [[0 for k in range(d)] for j in range(m)]

    for i in range(n):
        for k in range(d):
```

```
            X[i][k] = u[i * d + k] % 1000

    for j in range(m):
        for k in range(d):
            Y[j][k] = u[(n + j) * d + k] % 1000

    return X, Y

# Function to generate results for Q1
def q1(n, m, d):
    X, Y = gen_sets(d, n, m)
    print("(" + str(X[33][13]) + ", " + str(Y[123][43]) + ")")

q1(1000, 10000, 128)
q1(1000, 10000, 200)
q1(100, 10000, 128)
```

On appelle *plus proche voisin* de $\mathbf{y}_j$ dans $\mathcal{X}$ le vecteur $\mathbf{x}$ défini par :

$$\mathbf{x} = \arg\min_{\mathbf{x}_i \in \mathcal{X}} \left(\|\mathbf{x}_i - \mathbf{y}_j\|_2\right)^2 ,$$

c'est-à-dire le vecteur de $\mathcal{X}$ le plus proche au sens du carré de la distance Euclidienne du vecteur $\mathbf{y}_j$. Notons qu'il est possible que plusieurs $\mathbf{x}_i$ soient simultanément les plus proches voisins du vecteur $\mathbf{y}_j$. On choisira alors systématiquement celui d'indice le plus faible dans le reste de cet énoncé.
Cette notion est étendue à celle de k-ième plus proche voisin, en retirant de $\mathcal{X}$ un à un $k-1$ fois le plus proche voisin, puis en calculant le plus proche voisin sur la liste résultante.

Enfin, on ne cherchera pas à vérifier si tous les vecteurs $\mathbf{x}_i$ sont distincts deux à deux. Dans un tel cas, seul celui d'indice le plus faible pourra être considéré un plus proche voisin d'un vecteur donné.

2 Recherche naïve

Pour trouver les plus proches voisins d'un vecteur $\mathbf{y}_j$, on propose dans un premier temps de réaliser une recherche exhaustive. Elle consiste à calculer pour tout $0 \leq i < n$ les distances $\left(\|\mathbf{x}_i - \mathbf{y}_j\|_2\right)^2$ puis à identifier ensuite les plus proches voisins. On qualifie cet algorithme de naïf. On répète ce procédé indépendamment pour tous les vecteurs de $\mathcal{Y}$.

Question orale 1. *Estimer (en utilisant la notation $\mathcal{O}$) la complexité de cet algorithme en fonction des paramètres du problème (n, m et d).*

Cette première question orale n'a posé aucune difficulté à l'ensemble des candidats. On remarque immédiatement qu'on réalise m fois un calcul de n distances, chacune coûtant de l'ordre de d opérations. Soit une complexité totale en $\mathcal{O}(nmd)$.

Question 2. *Pour $n = 100\,000$, $m = 1$ et les valeurs de d suivantes, déterminer l'indice du plus proche voisin dans $\mathcal{X}$ de y_0 : a) $d = 1$, b) $d = 32$, c) $d = 128$.*

À condition d'avoir bien tabulé les valeurs du générateur pseudo-aléatoire, cette question ne pose pas de difficulté particulière. Elle avait également pour but de tester que les candidats rendaient bien le premier indice d'un plus proche voisin.

```python
# Distance computation
def distance(x, y):
    assert(len(x) == len(y))
    total = 0
    for i in range(len(x)):
        total += (x[i] - y[i]) ** 2
    return total

# Naive search
def naive_search(y, X):
    best_distance = len(y) * 1000000
    best_index = (-1)
    for i in range(len(X)):
        d = distance(y, X[i])
        if d < best_distance:
            best_distance = d
            best_index = i
    return best_index

# Function to generate results for Q2
def q2(d):
    X, Y = gen_sets(d, 100000, 1)
    print(naive_search(Y[0], X))

q2(1)
q2(32)
q2(128)
```

Question orale 2. *Justifier du fait que tout algorithme permettant de trouver les plus proches voisins de m vecteurs requêtes (avec $n \geq 2$) a une complexité en $\Omega(md)$ (on rappelle que pour deux suites u et v, $u = \Omega(v)$ signifie : $\exists n_0 \in \mathbb{N}, \exists M > 0, \forall n \geq n_0, u \geq Mv$).*

Nous avons été étonnés de constater que cette question a été d'une difficulté importante pour les candidats. Il semble que la notion de complexité soit mal comprise par beaucoup d'entre eux.

Pour d et m au moins égaux à 3, on peut choisir $\mathcal{X}$ comme contenant un premier vecteur entièrement à 0, et tous ses autres vecteurs exactement à 1, et $\mathcal{Y}$ comme ne contenant que des vecteurs nuls. Par

l'absurde, choisissons un algorithme dont la complexité est plus faible que $md/10$, alors cet algorithme ne consulte que 10% des coordonnées des vecteurs de $\mathcal{Y}$ au plus.

On peut alors choisir de construire $\mathcal{Y}'$ comme identique à $\mathcal{Y}$ pour les coordonnées consultées, et pour lesquels tous les bits sont inversés pour les autres coordonnées. Puisque notre algorithme ne consulte que 10% des coordonnées, il y a au moins un vecteur dans $\mathcal{Y}$ pour lequel au plus 10% des coordonnées sont visitées. Ce vecteur et son correspondant dans $\mathcal{Y}'$ ont donc des plus proches voisins distincts dans $\mathcal{X}$. Or, notre algorithme ne consultant que les parties communes entre ces vecteurs ne peut détecter cette différence.

On appelle *représentativité* de $\mathbf{x}_i$ dans $\mathcal{X}$ et pour $\mathcal{Y}$ le nombre de fois que $\mathbf{x}_i$ est le plus proche voisin d'un vecteur de $\mathcal{Y}$ (on rappelle pour la dernière fois que si un vecteur $\mathbf{y}_j$ est à distance minimale de plusieurs vecteurs dans $\mathcal{X}$, seul celui d'indice le plus faible est considéré).

Question 3. *Pour $n = 100$, $m = 10\,000$ et $d = 32$, calculer les représentativités de : a) $\mathbf{x}_{12}$, b) $\mathbf{x}_{34}$, c) $\mathbf{x}_{56}$.*

À nouveau, cette question ne posait pas de difficulté particulière.

```python
# Function to generate results for Q3
def q3(i):
    X, Y = gen_sets(32, 100, 10000)
    total = 0
    for j in range(len(Y)):
        if naive_search(Y[j],X) == i:
            total += 1
    print(total)

q3(12)
q3(34)
q3(56)
```

Soit $\mathbf{z}$ un vecteur de $\{0,\dots,999\}^d$. On note $r(\mathbf{z})$ la représentativité de $\mathbf{z}$ dans $\mathcal{X}_\mathbf{z} = [\mathbf{x}_0,\dots,\mathbf{x}_{n-1},\mathbf{z}]$ et pour $\mathcal{Y}$. En d'autres termes, on s'intéresse à la représentativité de $\mathbf{z}$ lorsqu'il est ajouté en tant que dernier élément (d'indice n) dans $\mathcal{X}$.

Question 4. Pour $d = 1$, *déterminer* $\max_{\mathbf{z}\in\{0,\dots,999\}^d} r(\mathbf{z})$ *pour les paramètres suivants : a)* $n = 3$, $m = 1000$, *b)* $n = 3$, $m = 10\,000$, *c)* $n = 3$, $m = 100\,000$.

Compte tenu des paramètres de la question, il était possible de la résoudre en calculant indépendamment pour chaque valeur de $\mathbf{z}$ la représentativité obtenue. Cela prenait toutefois un temps important. De façon plus maline, on attendait qu'un candidat remarque qu'à chaque vecteur $\mathbf{y}$ de $\mathcal{Y}$ pouvait être associé l'ensemble des vecteurs $\mathbf{z}$ qui seraient plus proches voisins de $\mathbf{y}$ dans $\mathcal{X}_\mathbf{z}$ en ne faisant qu'une seule fois le calcul du plus proche voisin.

```python
# Function to search maximum representativity for d = 1
def max_repr(X, Y):
    scores = [0] * 1000
```

```
    for j in range(len(Y)):
        i = naive_search(Y[j], X)
        dist = abs(Y[j][0] - X[i][0])
        for l in range(-dist + 1, dist):
            try:
                scores[l + Y[j][0]] += 1
            except:
                pass
    maximum = 0
    for l in range(1000):
        maximum = max(maximum, scores[l])
    return maximum

# Function to generate results for Q4
def q4(n, m):
    X, Y = gen_sets(1, n, m)
    print(max_repr(X, Y))

q4(3, 1000)
q4(3, 10000)
q4(3, 100000)
```

Les parties 4, 5 et 6 peuvent être résolues de façon indépendante.

3 Graphe des k plus proches voisins

On appelle *graphe des k plus proches voisins* associé à $\mathcal{X} = [\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}]$ le graphe $G = \langle \{0, \ldots, n-1\}, E \rangle$ où E est défini comme suit : $(u, v) \in E$ si et seulement si $\mathbf{x}_u$ est l'un des k plus proches voisins de $\mathbf{x}_v$ dans $\mathcal{X} - \{\mathbf{x}_v\}$ ou $\mathbf{x}_v$ est l'un des k plus proches voisins de $\mathbf{x}_u$ dans $\mathcal{X} - \{\mathbf{x}_u\}$. La notation $\mathcal{X} - \{\mathbf{x}_v\}$ désigne la suite obtenue en ignorant l'élément d'indice v.

Question orale 3. *Montrer qu'un graphe des k plus proches voisins est un graphe non orienté.*

Cette question n'a posé aucune difficulté. Elle avait pour principal objectif de guider la solution pour la question écrite suivante.

On remarque tout de suite que la définition de $(u, v) \in E$ est symétrique par rapport à u et v.

Question 5. *Pour chaque jeu de paramètres donné, trouver la plus petite valeur de k pour laquelle le graphe des k plus proches voisins associé à $\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}$ est connexe[a] : a)* $n = 100$, $d = 128$, *b)* $n = 200$, $d = 30$, *c)* $n = 300$, $d = 26$, *d)* $n = 1000$, $d = 1$.

a. On rappelle qu'un graphe connexe est un graphe pour lequel toute paire de sommets peut être reliée par un chemin.

Plusieurs solutions étaient envisageables. De façon à limiter la complexité, on attendait que les candidats factorisent un maximum de calculs. En résumé, la méthode attendue réalise trois étapes :

1. Une construction de la matrice des distances entre les vecteurs,
2. L'obtention de la liste triée des plus proches voisins pour chaque indice,
3. Une recherche dichotomique de la valeur k limite, en testant la connexité par un parcours de graphe à partir d'un sommet quelconque (exploitant le caractère non-orienté du graphe).

Cette question avait pour principal objectif de vérifier que les candidats étaient capables d'assembler plusieurs fonctions simples/vues en cours ensemble pour former un programme plus complexe.

```python
# Function to depth-first search a graph
def DFS(graph, v0):
    visited = [False for i in range(len(graph))]
    visited[v0] = True
    stack = [v0]
    while stack != []:
        v = stack.pop()
        for neighbor in graph[v]:
            if not(visited[neighbor]):
                visited[neighbor] = True
                stack.append(neighbor)
    return visited

# Function to assess if a graph is connected
def is_connected(graph):
    visited = DFS(graph, 0)
    connected = True
    for i in range(len(graph)):
        if not(visited[i]):
            connected = False
    return connected

# Function to generate a graph from neighbors and k
def graph_of_neighbors(neighbors, k):
    graph = [[] for i in range(len(neighbors))]
    for i in range(len(neighbors)):
        for ii in range(k):
            if i not in graph[neighbors[i][ii]]:
                graph[neighbors[i][ii]].append(i)
            if neighbors[i][ii] not in graph[i]:
                graph[i].append(neighbors[i][ii])
    return graph

```

```python
# Function to sort neighbors in order given a distance matrix
def neighbors(distances):
    neighbors = [[[0,0] for i in range(len(distances))] for ii in range(len(distances))]
    for i in range(len(distances)):
        for ii in range(len(distances)):
            if i == ii:
                neighbors[i][ii] = [ii, 1000000000000]
            else:
                neighbors[i][ii] = [ii, distances[i][ii]]
        neighbors[i] = sorted(neighbors[i], key = lambda elt: elt[1]*(len(distances)) + elt[0])
        for ii in range(len(distances)):
            neighbors[i][ii] = neighbors[i][ii][0]
    return neighbors

# Function to compute distances
def distances(X):
    return [[distance(X[i],X[ii]) for i in range(len(X))] for ii in range(len(X))]

# Function to generate answers to Q5
def q5(d, n):
    X, _ = gen_sets(d, n, 0)
    dists = distances(X)
    neighs = neighbors(dists)
    k_min = 0
    k_max = n - 1
    while k_min != k_max:
        k_middle = (k_min + k_max) // 2
        graph = graph_of_neighbors(neighs, k_middle)
        if is_connected(graph):
            k_max = k_middle
        else:
            k_min = k_middle + 1
    print(k_min)

q5(128, 100)
q5(30, 200)
q5(26, 300)
q5(1, 1000)
```

Question orale 4. *Présenter votre algorithme et donner sa complexité.*

Pour la méthode proposée, une analyse de complexité valide était :

1. Pour la génération des distances, on obtient $\mathcal{O}(n^2 d)$ (voir question orale 1),

2. Pour obtenir les vecteurs les plus proches triés, il est effectué n tris, chacun sur n valeurs, soit une complexité en $\mathcal{O}(n^2 \log(n))$,
3. Enfin, on effectue une recherche dichotomique sur un intervalle de diamètre $\mathcal{O}(n)$, soit $\mathcal{O}(\log(n))$ étapes, avec à chaque fois la construction du graphe en $\mathcal{O}(n^2)$ puis le test de connexité (parcours) en $\mathcal{O}(n^2)$.

Au final, on obtenait une complexité en $\mathcal{O}\left(n^2\left(d + \log(n)\right)\right)$. La plupart des candidats ont montré une complexité en $\mathcal{O}\left(n^2\left(d + n\right)\right)$, ce qui était également accepté.

On appelle *attracteur* un sommet du graphe des k plus proches voisins étant voisin de tous les autres sommets.

Question 6. *Pour les jeux de paramètres suivants, déterminer le plus petit entier k pour lequel le graphe des k plus proches voisins associé à $\mathcal{X}$ admet au moins un attracteur : a)* $n = 100$, $d = 128$, *b)* $n = 300$, $d = 300$, *c)* $n = 500$, $d = 1000$.

Ici aussi, on attendait des candidats une factorisation des opérations à effectuer.

Il s'agissait donc de chercher par dichotomie la valeur de k limite. Alternativement, il était possible de chercher une valeur de k linéairement en partant de $n - 1$, en retirant les non-attracteurs lorsque k diminuait, jusqu'à trouver la valeur limite. C'est cette méthode qui est présentée dans le code suivant.

```python
# Function to generate answers to Q6
def q6(d, n):
    X, _ = gen_sets(d, n, 0)
    dists = distances(X)
    neighs = neighbors(dists)
    attractors = list(range(n))
    k = n - 1
    while len(attractors) > 0:
        k -= 1
        for i in attractors.copy():
            for ii in range(n):
                if i != ii and not(i in neighs[ii][:k] or ii in neighs[i][:k]):
                    attractors.remove(i)
                    break
    print(k + 1)

q6(128, 100)
q6(300, 300)
q6(1000, 500)
```

Question orale 5. *Pour $d = n$ non triviaux, donner un exemple de $\mathcal{X}$ pour lequel le plus petit entier k demandé à la question 6 est 1. Même question avec le plus petit entier k qui vaut $n - 1$.*

Beaucoup de candidats ont pensé à proposer une liste de vecteurs $\mathcal{X}$ tous égaux pour le premier cas. Alternativement, on pouvait penser à prendre le vecteur nul et des vecteurs distincts de la base canonique.

Pour le second cas, une solution possible consistait à prendre des vecteurs racines n-ièmes de l'unité (les autres coordonnées pouvant être choisies nulles).

On appelle *clique* dans un graphe $G = \langle V, E \rangle$ un sous-ensemble S de sommets tels que $\forall u, v \in S, u \neq v \Rightarrow (u, v) \in E$.

Question 7. *Pour chacun des graphes des k plus proches voisins obtenus à partir de $\mathcal{X}$ avec les paramètres suivants, indiquer la cardinalité de sa plus grande clique : a)* $n = 100, d = 128, k = 15$, *b)* $n = 100, d = 200, k = 20$, *c)* $n = 100, d = 222, k = 30$.

La solution choisie par l'ensemble des candidats ayant résolu cette question était de procéder par extension, un sous-ensemble d'une clique étant une clique. Dans l'algorithme suivant, on garde des listes triées pour représenter chaque sous-clique.

```python
# Maximum clique search
def largest_clique(graph):
    sub_cliques = [([],-1)] # we store subcliques and their maximum element
    size = 0
    while len(sub_cliques) > 0:
        new_cliques = []
        for (clique, max_elt) in sub_cliques:
            for i in range(max_elt + 1, len(graph)):
                is_extensible = True
                for elt in clique:
                    if not (elt in graph[i]):
                        is_extensible = False
                        break
                if is_extensible:
                    new_cliques.append((clique.copy() + [i], i))
        sub_cliques = new_cliques
        size += 1
    return size - 1

# Function to generate answers to Q7
def q7(d, n, k):
    X, _ = gen_sets(d, n, 0)
    dists = distances(X)
    neighs = neighbors(dists)
    graph = graph_of_neighbors(neighs, k)
    print(largest_clique(graph))

q7(128, 100, 15)
```

```
q7(200, 100, 20)
q7(222, 100, 30)
```

Question orale 6. *Présenter votre algorithme et démontrer qu'il rend un résultat correct.*

Il était attendu des candidats qu'ils mentionnent que $\{v_1, v_2, \ldots, v_\ell\}$ étant une clique maximale, tout sous-ensemble $\{v_1, v_2, \ldots, v_p\}$ est également une clique, avec le cas final de l'ensemble vide. Ainsi, par descente, cette clique maximale sera obtenue par l'algorithme.

4 Faible dimensionalité

Question 8. *Dans le cas où $d = 2$, déterminer l'indice de l'élément de $\mathcal{X}$ de représentativité maximale (en cas d'égalité, on donnera uniquement le plus petit indice) dans les cas suivants : a) $n = 100\,000$, $m = 10\,000$, b) $n = 1\,000\,000$, $m = 100\,000$, c) $n = 1\,000\,000$, $m = 1\,000\,000$.*

Le titre de cette partie était très important pour comprendre ce qui était attendu. On observe qu'en dimension 2, il est possible d'associer à chaque couple d'entiers (a, b) avec $0 \leq a, b < 1000$, l'indice minimal d'un vecteur dans $\mathcal{X}$ égal à (a, b), s'il en existe un. Rechercher le vecteur le plus proche d'un vecteur requête peut alors s'effectuer en recherchant le couple (a, b) le plus proche ayant une correspondance dans $\mathcal{X}$.

```
# Function to find inverse map from 2D vectors to indexes in X
def inverse_map(X):
    imap = [[(-1) for i in range(1000)] for ii in range(1000)]
    for i in range(len(X)):
        if imap[X[i][0]][X[i][1]] == -1:
            imap[X[i][0]][X[i][1]] = i
    return imap

import math

# Function to generate answers to Q8
def q8(n, m):
    X, Y = gen_sets(2, n, m)
    imap = inverse_map(X)
    scores = [0 for i in range(n)]
    for j in range(m):
        neighbors = []
        dist = 0
        while neighbors == []:
            rg = int(math.sqrt(dist))
            for a in range(-rg, rg + 1):
                b = int(math.sqrt(dist - (a ** 2)))
```

```
                if a ** 2 + b ** 2 == dist:
                    try:
                        if imap[Y[j][0] + a][Y[j][1] + b] != -1:
                            neighbors.append(imap[Y[j][0] + a][Y[j][1] + b])
                    except:
                        pass
                    try:
                        if imap[Y[j][0] + a][Y[j][1] - b] != -1:
                            neighbors.append(imap[Y[j][0] + a][Y[j][1] - b])
                    except:
                        pass
            dist = (int(math.sqrt(dist)) + 1) ** 2
        scores[sorted(neighbors)[0]] += 1
    maximum = 0
    index = (-1)
    for i in range(len(scores)):
        if scores[i] > maximum:
            maximum = scores[i]
            index = i
    print(index)

q8(100000, 10000)
q8(1000000, 100000)
q8(1000000, 1000000)
```

Question orale 7. *Présenter votre algorithme et donner sa complexité.*

Puisqu'on s'intéresse ici uniquement au cas où d est petit, on prête attention uniquement au comportement en fonction de n et m. On remarque que la création de la carte pour les vecteurs de $\mathcal{X}$ est en $\mathcal{O}(n)$ alors que le calcul des plus proches voisins est en $\mathcal{O}(m)$. On obtient donc une complexité en $\mathcal{O}(n+m)$. Aux étudiants qui mettaient cette quantité en regard de celle obtenue en question orale 2 pour montrer que cet algorithme est de complexité optimale dans certains cas, nous offrions un bonus de points.

5 Vecteurs binaires

À partir de maintenant et pour le reste de l'énoncé, on remplacera les vecteurs générés habituellement en appliquant $x \mapsto x \pmod 2$ à toutes les coordonnées. Ainsi :

$$\forall 0 \leq k < d, \begin{cases} \mathbf{x}_i[k] = u_{id+k} \mod 2, \\ \mathbf{y}_j[k] = u_{(n+j)d+k} \mod 2, \end{cases}$$

Question 9. *Déterminer l'indice de l'élément de $\mathcal{X}$ de représentativité maximale (en cas d'égalité, on donnera uniquement le plus petit indice) dans les cas suivants : a)* $n = 1\,000$, $d = 10$, $m = 1\,000\,000$, *b)* $n = 10\,000$, $d = 10$, $m = 1\,000\,000$, *c)* $n = 1\,000\,000$, $d = 20$, $m = 1\,000\,000$.

Il s'agit d'une variation de la question précédente demandant un peu plus de travail car ici la carte inversée est de taille variable.

```python
# Vec to index
def vec_to_index(x):
    value = 0
    for i in range(len(x)):
        value += (2 ** i) * x[i]
    return value

# Construction of inverse map
def inverse_bool_map(X):
    D = 2 ** (len(X[0]))
    res = [(-1) for i in range(D)]
    for i in range(len(X)):
        index = vec_to_index(X[i])
        if res[index] == -1:
            res[index] = i
    return res

# Function to generate boolean sets X and Y given d, n and m
def gen_boolean_sets(d, n, m):
    X = [[0 for k in range(d)] for i in range(n)]
    Y = [[0 for k in range(d)] for j in range(m)]

    for i in range(n):
        for k in range(d):
            X[i][k] = u[i * d + k] % 2

    for j in range(m):
        for k in range(d):
```

```
            Y[j][k] = u[(n + j) * d + k] % 2

    return X, Y

# Function to generate answers to Q9
def q9(d, n, m):
    X, Y = gen_boolean_sets(d, n, m)
    imap = inverse_bool_map(X)
    scores = [0 for i in range(n)]
    for j in range(m):
        neighbors = []
        modifs = [[]]
        while neighbors == []:
            for modif in modifs:
                value = Y[j].copy()
                for elt in modif:
                    value[elt] = 1 - value[elt]
                value = vec_to_index(value)
                if imap[value] != -1:
                    neighbors.append(imap[value])
            new_modifs = []
            for modif in modifs:
                try:
                    val_min = modif[-1] + 1
                except:
                    val_min = 0
                for k in range(val_min, d):
                    new_modifs.append(modif + [k])
            modifs = new_modifs
        scores[sorted(neighbors)[0]] += 1
    maximum = 0
    index = (-1)
    for i in range(len(scores)):
        if scores[i] > maximum:
            maximum = scores[i]
            index = i
    print(index)


q9(10, 1000, 1000000)
q9(10, 10000, 1000000)
q9(20, 1000000, 1000000)
```

Question orale 8. *Présenter votre algorithme.*

Proposition de solution pour le sujet : sous-suites

On commence par une fonction d'affichage

```
In [1]: u0 = 42

        def qprint(q, i, s):
            print(str(q) + "." + chr(ord('a') + i) + " " + str(s))
```

Question 1

On prend bien soin de calculer u une fois pour toute dans un tableau de taille suffisante pour tout le sujet.

```
In [2]: m = (1 << 31) - 1
        nmax = 1 << 20

        def init(u0):
            u = [u0]
            for i in range(1, nmax):
                u.append((16807*u[i-1] + 17) % m)
            return u

        u = init(u0)

        ################################
        def q1():
            for a, n in enumerate([16,
                                   1024,
                                   32768]):
                qprint(1, a, u[n] % 101)

        q1()
```

```
1.a 62
1.b 0
1.c 35
```

Question 2

```
In [3]: def omega(l, n):
            r = [ u[n] % l for n in range(n, n+l) ]
```

```
        return r

    def Omega(s):
        return [ s[n] for n in s ]

    ################################
    def q2():
        for a, (l, n) in enumerate([(100, 100),
                                    (1000, 200),
                                    (100982, 300)]):
            s1 = omega(l, n)
            s2 = Omega(s1)
            qprint(2, a, sum(s2) % 101)

    q2()
```

```
2.a 77
2.b 96
2.c 28
```

Question 3

Comme suggéré par la question d'oral 1, la longueur de la plus longue sous-suite strictement croissante est la même que la taille de la couverture minimale. Donc il suffit d'appliquer l'algorithme calculant la couverture gloutonne et, par application de la question d'oral 2, on en déduit la longueur souhaitée.

Une implémentation naïve de l'algorithme glouton est en O(l^2). Les sous-suites étant décroissantes, on remarque qu'il suffit de maintenir le dernier élément de chaque sous-suite pour déterminer k en O(log(l)) étapes. La complexité est alors O(l log(l)).

```
In [4]: def recherche_dichotomique(a, x, lo = 0):
            hi = len(a)
            while lo < hi:
                mid = (lo+hi)//2
                if a[mid] < x: lo = mid+1
                else: hi = mid
            return lo

        def insertion(g, v):
            k = recherche_dichotomique(g, v)
            if k == len(g):
                return -1
            else:
                return k

        def couverture(s):
            g = [ s[0] ]
```

```
            for i in range(1, len(s)):
                k = insertion(g, s[i])
                if k != -1:
                    g[k] = s[i]
                else:
                    g.append(s[i])
            return len(g)

        ################################
        def q3():
            for a, (l, n) in enumerate([(10135, 201),
                                        (20562, 224),
                                        (100123, 245),
                                        (110428, 289)]):
                s = omega(l, n)
                qprint(3, a, couverture(s) % 101)

        q3()
```

```
3.a 95
3.b 70
3.c 21
3.d 37
```

Question 4

```
In [5]: def fusion(s):
            d = {}

            for i in range(len(s)):
                if not s[i][0] in d:
                    d[s[i][0]] = []
                d[s[i][0]].append(s[i][1])

            return d

        def Pi(s1, s2):
            s2p = [ (x, i) for i, x in enumerate(s2) ]
            s2p.sort(key=lambda xy: (xy[0], -xy[1]))
            tab = fusion(s2p)

            res = []
            for x in s1:
                v = tab.get(x, [])
                res.append(v)

            return res
```

```
        ################################
        def q4():
            for a, (l, n) in enumerate([(10, 505),
                                        (100, 514),
                                        (1000, 523),
                                        (10000, 540)]):
                s1 = omega(l, n)
                s2 = Omega(s1)
                qprint(4, a, len(sum(Pi(s1, s2), [])))

        q4()
```

```
4.a 15
4.b 228
4.c 2041
4.d 19825
```

Question 5

On obtient |LCS(s1,s2)| en cherchant la plus grande valeur de k pour laquelle sp(l, k) est défini.

Pour calculer sp($-$, =) on utilise corresp($-$, =, $\equiv$). En effet, au lieu de parcourir tous les j pour ne conserver que ceux qui vérifient s1(i $-$ 1) = s2(j $-$ 1), il suffit de parcourir corresp(s1, i-1, s2).

On peut pré-calculer corresp($-$, =, $\equiv$) en O(l log(l)) en faisant un tri lexicographique sur les paires "(symbole, position)" de s2 suivi d'une agrégation des positions selon les symboles (en un parcours de la liste triée, soit O(l)). On fait ensuite l recherches dichotomiques (chacune en O(log(l))) pour associer à chaque indice i de s1 l'ensemble des positions correspondantes dans s2.

Le calcul de sp(i+1, $-$) demande, pour chaque position i de s1 et pour chaque indice j de corresp(s1, i, s2), une recherche dans sp(i, $-$) qui est strictement croissant et de taille O(l), soit O(r(s1,s2) log(l)) opérations. La complexité temporelle globale est donc en O((l + r(s1,s2)) log(l)).

```
In [6]: def SP(s1, s2, imax):
            l = len(s2)

            infini = l+1

            pi = Pi(s1, s2)

            sp = [ infini ] * (l+1)
            sp[0] = 0

            for ip, x in enumerate(s1):
                i = ip + 1
                if i > imax:
                    break
                spn = sp
                for jp in pi[i-1]:
```

```
                    j = jp + 1
                    k = recherche_dichotomique(sp, j)
                    if k == 0 or k == len(sp):
                        continue
                    spn[k] = j
                sp = spn

            return sp

        def maxSP(sp):
            for k in reversed(range(len(sp))):
                if sp[k] < len(sp):
                    return k

        ################################
        def q5():
            for a, (l, n, i) in enumerate([(15, 601, 5),
                                           (400986, 657, 102),
                                           (500431, 678, 10502)]):
                s1 = omega(l, n)
                s2 = Omega(s1)
                sp = SP(s1, s2, i)
                qprint(5, a, maxSP(sp) % 101)

        q5()
```

```
5.a 3
5.b 21
5.c 79
```

Question 6

```
In [7]: def SPfinal(s1, s2):
            return SP(s1, s2, len(s1) + 1)

        def LCS(s1, s2):
            return maxSP(SPfinal(s1, s2))

        ################################
        def q6():
            for a, (l, n) in enumerate([(305412, 732),
                                        (700320, 745),
                                        (1000812, 763)]):
                s1 = omega(l, n)
                s2 = Omega(s1)
                qprint(6, a, LCS(s1, s2) % 101)
```

```
q6()
```

```
6.a 33
6.b 16
6.c 86
```

Question 7

Il s'agit ici de généraliser l'approche de la question 3.

Il s'agit de généraliser l'algorithme de couverture gloutonne en traitant des paires "(valeur, poids)" selon l'ordre lexicographique. Comme précédemment, on trouve un antécédent de façon gloutonne en choisissant la plus grande valeur inférieure strictement à celle que l'on souhaite insérer. Ensuite, on recouvre toutes les paires de poids inférieur à celui que l'on souhaite insérer.

```
In [8]: def w(s):
            w = [ 1 for _ in s ]
            for i, x in enumerate(s):
                for j in range(0, i):
                    if s[j] == s[i]:
                        w[i] = 3
                        break
            return w

        def cherche_minW(g, v):
            for k in range(len(g)):
                if g[k][0] >= v:
                    return k
            return len(g)

        def couvertureW(s, w):
            g = [ (-1, 0) ]
            for i in range(len(s)):
                k_start = cherche_minW(g, s[i])
                wi = g[k_start-1][1] + w[i]
                k = k_start
                while k < len(g) and wi >= g[k][1]:
                    k = k+1
                del g[k_start:k]
                g.insert(k_start, (s[i], wi))
            return g

        def his(g):
            return max([w for _, w in g])

        ################################
        def q7():
            for a, (l, n) in enumerate([(10142, 401),
```

```
                                        (20540, 428),
                                        (30489, 465)]):
                s = omega(l, n)
                weight = w(s)
                qprint(7, a, his(couvertureW(s, weight)) % 101)

        q7()
```

```
7.a 85
7.b 52
7.c 84
```

Question 8

Il s'agit de combiner la généralisation de la couverture gloutonne et le calcul de la plus longue sous-suite à partir de sa réduction au problème de la plus longue sous-suite croissante.

À nouveau, on utilise corresp($-$, $=$, $\equiv$) pour accélérer le parcours de tous les j vérifiant s1(i-1) = s2(j-1). La complexité du pré-calcul reste en O(l log(l)).

Le calcul de la HCS demande, à nouveau, pour chaque position i de s1 et pour chaque indice j de corresp(s1, i, s2), une recherche dans un tableau trié soit O(r(s1,s2) log(l)) opérations. La complexité totale est O((l + r(s1,s2)) log(l)).

```
In [9]: def v(s1, s2, i, j):
            len(s1) - abs(i - j)

        def corresp1(s1, i, s2):
            c = []
            w = []
            for j in reversed(range(len(s2))):
                if s1[i] == s2[j]:
                    c.append(j)
                    w.append(len(s1) - abs(i - j))
            return (c, w)

        def PiW(s1, s2):
            pi = []
            wi = []
            for i in range(len(s1)):
                (c, w) = corresp1(s1, i, s2)
                pi.extend(c)
                wi.extend(w)
            return pi, wi

        def hcs(s1, s2):
            c, w = PiW(s1, s2)
            g = couvertureW(c, w)
            return his(g)
```

```
        ################################
        def q8():
            for a, (l, n) in enumerate([(10156, 809),
                                        (15129, 826),
                                        (17845, 869)]):
                s1 = omega(l, n)
                s2 = Omega(s1)
                qprint(8, a, hcs(s1, s2) % 101)

        q8()
```

```
8.a 26
8.b 9
8.c 66
```

Question 9

```
In [10]: def couvertureWL(s, w):
             g = [ (-1, 0, 0) ]
             for i in range(0, len(s)):
                 k_start = cherche_minW(g, s[i])
                 wi = g[k_start-1][1] + w[i]
                 li = g[k_start-1][2] + 1
                 k = k_start
                 while k < len(g) and wi >= g[k][1] and (wi != g[k][1] or li <= g[k][2]):
                     k = k+1
                 del g[k_start:k]
                 g.insert(k_start, (s[i], wi, li))
             return g

         def his_minlen(g):
             max_w = max([ w for _, w, _ in g ])
             return min([ l for _, w, l in g if w == max_w ])

         def lhis(s, w):
             return his_minlen(couvertureWL(s, w))

         ################################
         def q9():
             for a, (l, n) in enumerate([(10378, 801),
                                         (20432, 832),
                                         (30654, 827)]):
                 s = omega(l, n)
                 weight = w(s)
                 qprint(9, a, lhis(s, weight) % 101)
```

```
q9()
```

```
9.a 63
9.b 42
9.c 99
```

Question 10

```
In [11]: def lhcs(s1, s2):
             c, w = PiW(s1, s2)
             g = couvertureWL(c, w)
             return his_minlen(g)

         ################################
         def q10():
             for a, (l, n) in enumerate([(10289, 804),
                                         (14912, 823),
                                         (18142, 892)]):
                 s1 = omega(l, n)
                 s2 = Omega(s1)
                 qprint(10, a, lhcs(s1, s2) % 101)

         q10()
```

```
10.a 81
10.b 33
10.c 61
```