Composition d'Informatique (2 heures)
Filières MP et PC
Concours 2013

Rapport de M. Didier CASSEREAU, Étienne LOZES, et Dominique ROSSIN, correcteurs

1 Statistiques

A titre de rappel, cette épreuve n'est corrigée que pour les candidats admissibles. Cette année encore, les copies de la filière MP et celles de la fillière PC ont été confiées à des correcteurs différents, et ont ainsi fait l'objet de deux barèmes distincts.

1.1 Statistiques de la filière MP

Cette année le nombre total de candidats admissibles dans cette filière est de 282. La note moyenne est de 12,82 avec un écart-type de 2,85. Les tableaux ci-dessous donnent la répartition détaillée des notes par série, ainsi que la synthèse calculée sur l'ensemble des copies corrigées. La note minimale est de 2/20 et la note maximale 19/20. Une copie a obtenu la note éliminatoire de 2/20. Trois copies n'ont pas été notées, les candidat(e)s ne s'étant pas présenté à l'épreuve.

	Série 1		Série 2		Série 3		Série 4		**Synthèse**	
$0 \leq N < 4$	1	1.3%	1	1.2%	0	0.0%	0	0.0%	**2**	**0,7%**
$4 \leq N < 8$	0	0.0%	5	6.0%	2	3.2%	0	0.0%	**7**	**2,5%**
$8 \leq N < 12$	26	32.9%	28	33.3%	17	27.4%	21	36.8%	**92**	**32,6 %**
$12 \leq N < 16$	39	49.4%	41	48.8%	34	54.8%	32	56.1%	**146**	**51,8%**
$16 \leq N \leq 20$	13	16.5%	9	10.7%	9	14.5%	4	7.0%	**35**	**12,4%**
Total	79	100,0%	84	100,0%	62	100,0%	57	100,0%	**282**	**100,0%**
Epreuve complète*	7	8,9%	7	8,3%	10	16,1%	8	14,0%	**32**	**11,3%**

* *Epreuve complète* signifie ici que le candidat a abordé toutes les questions de l'énoncé et obtenu une note non nulle à chacune des 14 questions.

	Série 1	Série 2	Série 3	Série 4	**Synthèse**
Nombre de copies	79	84	62	57	**282**
Note minimale	2,8	2,0	4,6	8,1	**2,0**
Note maximale	18,7	18,5	19,0	18,0	**19,0**
Note moyenne	12,89	12,62	13,05	12,76	**12,82**
Ecart-type	2,97	3,02	2,89	2,40	**2,85**

Le langage de programmation choisi par les candidats est largement dominé par Maple (qui est majoritairement bien orthographié, avec encore quelques candidat(e)s qui s'obstinent à composer en Mapple) avec 60,6% des copies, suivi ensuite par Caml (27,3% des copies) et C/C++ (4,6% des copies). On trouve enfin quelques copies en Python (2,8%), Java (2,1%), Pascal (0,7%) et autres (1,8%, incluant par exemple Mathematica). Le tableau ci-dessous illustre cette répartition des langages choisis par les candidats.

Maple	Caml	C/C++	Python	Pascal	Java	Autres	**Total**
171	77	13	8	2	6	5	**282**
60,6%	27,3%	4,6%	2,8%	0,7%	2,1%	1,8%	**100,0%**

Pour diverses raisons logistiques, les corrections ont été faites en deux vagues. La première vague incluait uniquement les copies des candidat(e)s admissibles à l'X, la seconde vague incluait quant à elle les candidat(e)s admissibles à l'ENS.
Cette correction a fait clairement apparaître une différence majeure par rapport au langage de composition utilisé par les candidat(e)s. Pour ce qui concerne la première vague, la grande majorité des copies a été rédigée en langage Maple, aucune copie n'a été faite en langage Caml. Pour la seconde vague, c'est exactement le contraire, avec une majorité écrasante de copies réalisées en Caml.
Un tel effet a déjà été observé l'an dernier, il est clairement confirmé de manière très forte cette année. Nous supposons que cette observation est liée au profil des classes préparatoires visant l'ENS, par opposition à celles visant l'X.

1.2 Statistiques pour la filière PC

Cette année le nombre total de candidats admissibles dans cette filière est de 520 . La note moyenne est de 9,2 avec un écart-type de 3,1. La note minimale est de 0 et la note maximale de 19,1.

	Série 1		Série 2		Série 3		Série 4		**Synthèse**	
$0 \leq N \leq 4$	10	6%	6	4%	8	6%	3	2%	**27**	**5%**
$4 \leq N \leq 8$	40	27%	44	32%	29	24%	31	26%	**144**	**27%**
$8 \leq N \leq 12$	74	50%	63	46%	56	47%	66	55%	**259**	**49%**
$12 \leq N \leq 16$	22	14%	21	15%	26	21%	17	14%	**86**	**16%**
$16 \leq N \leq 20$	1	0%	2	1%	0	0%	1	0%	**4**	**0%**
Total	147	100%	136	100%	119	100%	118	100%	**520**	**100%**

	Série 1	Série 2	Série 3	Série 4	**Synthèse**
Nombre de copies	147	136	119	118	**520**
Note minimale	0.0	0.5	2.2	1.4	**0.0**
Note maximale	17.1	19.1	15.1	16.1	**19.1**
Note moyenne	9.1	9.1	9.2	9.3	**9.2**
Ecart type	3.1	3.3	3.1	2.8	**3.1**

Le langage de programmation choisi par les candidats est largement dominé par Maple avec 94 % des copies, avec quelques rares copies en Python (5 %), ou encore Mathematica (1%).

2 Commentaires

Cette année le sujet portait sur le problème de la recherche des points fixes d'une fonction discrète $[0,n[\to [0,n[$. Le problème comportait 2 parties :

- la première partie, assez générale, permettait aux candidats de mettre en place les fonctions de base, et d'appliquer ces fonctions à la problématique de détermination d'un point fixe ; cette partie se terminait avec l'écriture d'algorithmes rapides et efficaces (en nombre d'opérations) pour ce genre de problème,
- la deuxième partie consistait à analyser spécifiquement le cas particulier des fonctions croissantes.

Les candidats étaient invités à passer outre certaines contraintes liées au langage de programmation (l'indexation des tableaux qui commence à 0 et non à 1). Cette directive a été largement respectée par la grande majorité des candidats. Cependant certains candidats n'ont pas suivi cette recommandation, avec généralement des conséquences néfastes, parmi lesquelles des erreurs dans les indices.
Nous recommandons donc explicitement aux candidats de respecter ce genre de directive donnée dans l'énoncé.

L'objectif de cette épreuve est d'évaluer la capacité des candidats à écrire un code informatique permettant de résoudre un problème algorithmique donné. Il est important de proscrire tout ce qui relève du calcul formel ; dans le cas contraire le candidat prend le risque de se voir sanctionné par rapport à d'autres copies dans lesquelles on peut trouver un code complet et juste.
Nous invitons en particulier les candidats composant dans des langages tels que Mathematica à être très vigilants sur ce point. Nous avons déjà mentionné ce genre de réserves dans les rapports des années précédentes, et nous tenons à les rappeler une fois encore cette année.

L'évaluation d'un code informatique repose sur plusieurs éléments :

- **correction** : l'algorithme sous-jacent au programme doit donner le résultat attendu, le cas échéant avec la complexité attendue ;
- **sobriété** : le code doit être aussi dépouillé que possible, de sorte que l'algorithme mis en œuvre soit facilement compréhensible ;
- **lisibilité** : un soin particulier doit être apporté dans la manière de présenter le code (indentation des boucles et tests, passages à la ligne, ...), les noms de variables et de fonctions doivent signifier ce qu'ils représentent, et les commentaires explicatifs ne doivent être introduits que s'il semble vraiment difficile au candidat de faire comprendre le principe de l'algorithme qu'il a mis en oeuvre par la lecture du code seul ;
- **efficacité** : la solution mise en oeuvre doit être efficace, et entre deux solutions de complexité conceptuelles comparables, on préfèrera toujours la plus efficace.

Quelques remarques générales à la lecture des codes :

- en cas de passage illisible ou incomplet, ce n'est pas au correcteur d'évaluer ce que le candidat *aurait pu répondre* ; il n'est pris en compte que ce qui est inscrit lisiblement sur la copie !
- cette lisibilité est d'autant plus importante pour les codes que le candidat doit écrire ; le langage informatique étant un langage structuré, la copie se doit de refléter cette structure et la logique du langage, ainsi que la logique de l'algorithme implémenté,
- on trouve énormément d'instructions totalement inutiles, dans le genre `x=x`, ou des clauses `else` rattachées à un test `if` et qui ne font rien ; ces instructions ne servent à rien sinon à obscurcir le code, il serait préférable de les éviter,
- on observe souvent un usage abusif de la récursivité; dans certains cas, elle est tout à fait adaptée, mais la plupart du temps, une simple boucle for ou while est plus proche de l'algorithmique sous-jacent. Il appartient au candidat de choisir l'approche qui rendra le mieux compte de l'algorithme, et de ne préférer une autre approche que dans la mesure où elle conduit à un programme plus efficace.
- en complément de la remarque précédente, beaucoup de candidats définissent des fonctions intermédiaires (c'est plutôt bien), mais elles s'appellent toutes `aux` (et là c'est moins bien) ; cela ne coûte rien de choisir un nom plus représentatif de la tâche exécutée par la fonction, et cela aide beaucoup à comprendre le code ! cette remarque et la précédente visent essentiellement les candidats composant en Caml,
- on ne demande pas aux fonctions d'afficher vrai ou faux, ou plus généralement d'afficher le résultat obtenu ; les fonctions doivent résoudre le problème posé et retourner un résultat,
- on voit souvent des solutions qui sont justes, mais difficiles à lire et à comprendre ; clairement une telle situation doit être évitée,
- certaines notations n'ont pas de sens dans un langage informatique, telles que par exemple `t'`, θ, ou encore les accents dans les noms de variables ou fonctions,

- il faut faire attention de ne pas manipuler des variables qui ne sont pas initialisées, par exemple dans une fonction qui cherche à évaluer la valeur maximale d'un tableau,
- les éléments de structuration du code (boucles ou tests) ont un début et une fin qui sont bien marqués par les règles syntaxiques du langage, il est impératif que les candidats veillent à spécifier ces mots-clés de manière précise et systématique ; dans le cas contraire, on a confusion totale sur ce qu'est supposé faire le code, et cette confusion n'est jamais à l'avantage du candidat,
- la syntaxe est parfois très approximative, avec un flou volontaire ou non ; c'est en particulier souvent le cas des candidats qui composent en langage C/C++,
- certains candidats n'utilisent que des boucles `while`, alors que dans certaines circonstances la boucle `for` est plus intuitive et plus simple,
- de manière générale, on peut souvent considérer que les différentes questions forment une progression, il ne faut donc pas hésiter à utiliser les fonctions écrites dans les questions précédentes, plutôt que de tout refaire,
- il est impératif d'écrire les codes de manière complète et précise ; on ne peut absolument pas accepter des écritures partielles, par exemple en omettant les paramètres que l'on transmet lors d'un appel de fonction, parce que c'est peut-être justement là que l'on pourra faire la différence entre un code qui est correct et un code qui ne l'est pas,
- il faut éviter d'appeler la même fonction de manière itérative, par exemple dans une boucle ; il est préférable d'appeler la fonction une seule fois, stocker le résultat en mémoire, et ensuite réutiliser cette variable,
- dans une boucle, par exemple indicée par une variable entière `i`, on ne peut pas manipuler une variable nommée `ti` en espérant que le nom de cette variable va suivre l'évolution de `i` ; cette erreur est assez fréquente et aboutit inévitablement à un code qui n'a pas grand sens et qui n'a surtout aucune chance de fonctionner,
- une erreur fréquente consiste à incrémenter le compteur d'une boucle `for` ; or le mécanisme de la boucle implique déjà cette incrémentation, de sorte que cela conduit à incrémenter deux fois, ce qui n'est en général pas ce que l'on cherche à faire,
- certains opérateurs logiques sont utilisés de manière un peu approximative, comme par exemple $><$ à la place de $<>$ ou $!=$, ou encore $=<$ à la place de $<=$,
- petit rappel à l'intention des utilisateurs de Python : l'instruction `range(a,b)` inclut `a` mais pas `b`.

Les copies rédigées en langage Caml sont souvent plus difficiles à lire en raison de la présentation du code lui-même:

- la programmation récursive est utilisée sans discernement là où de simples boucles sont largement suffisantes, plus claires et plus efficaces; la récursivité devrait être limitée aux situations dont la logique est principalement récurrente ; dans le cas contraire, les candidat(e)s prennent le risque de codes difficiles à comprendre et donc à évaluer,
- la définition de fonctions imbriquées contribue largement à obscurcir les codes,
- toutes les copies rédigées en langage Caml comportent des fonctions nommées `aux`, voire `aux1` ou `aux2` : nous encourageons vivement les candidat(e)s à prendre l'habitude de nommer leurs fonctions de manière plus pertinente,
- nous invitons fortement tous les candidats à être très vigilants sur la manière dont les codes sont présentés et écrits ; ceci vaut pour Caml, mais en réalité pour tous les langages de programmation.

3 Commentaires détaillés

Le tableau ci-dessous donne un aperçu de la difficulté par question; le nombre d'étoile donne une estimation cette difficulté, évaluée par la réussite des candidats (assez dissemblable sur certaines questions entre filières MP et PC):

- une étoile: question majoritairement réussie
- deux étoiles: beaucoup d'erreurs, mais une majorité de bonnes copies
- trois étoiles: question peu réussie ou peu traitée

Question	Description	Difficulté	Question	Description	Difficulté
1	`admet_point_fixe`	*	8	`est_croissante`	**
2	`nb_points_fixes`	*	9	`point_fixe`	***
3	`itere`	*	10	terminaison	**
4	`nb_points_fixes_iteres`	*	11	plus petit point fixe	*
5	`admet_attracteur_principal`	***	12	pgcd	*
6	`temps_de_convergence`	*	13	`pgcd_points_fixes`	***
7	`temps_de_convergence_max`	***	14	complexité logarithmique	***

Question 1 :
Cette question consiste à déterminer si une fonction, représentée par le tableau de ses valeurs, comporte un point fixe. Cette question ne présente pas de difficulté particulière, elle peut être traitée par une simple boucle `while` ou `for`. De nombreuses copies rédigées en langage Caml utilisent ici une fonction récursive, ce qui ne semble pas très naturel compte tenu du problème posé. A noter que la valeur de retour de cette fonction est connue dès qu'un point fixe est trouvé ; dans ce cas il n'est alors pas nécessaire de poursuivre l'exécution de la boucle.

Question 2 :
Cette question n'est finalement qu'une petite variante de la question précédente, dans la mesure où il est question ici de dénombrer les points fixes. A la différence de la fonction précédente, la boucle doit être parcourue dans son intégralité, avec simple incrémentation d'un compteur dès lors que l'on trouve un point fixe.

Question 3 :
Cette question consiste à itérer la fonction pour une valeur initiale donnée. Là encore il n'y a pas de véritable difficulté, il faut juste veiller à itérer un nombre correct de fois (les erreurs fréquentes consistent soit à oublier une itération, soit à en faire une de trop). Dans la mesure où l'on cherche ici uniquement à itérer pour une valeur initiale particulière donnée, il est inutile d'itérer l'ensemble du tableau, pour finalement ne conserver que la valeur demandée. Une autre erreur classique consiste à itérer une instruction du genre `y=t[x]` (cette instruction ne progresse pas, même exécutée en boucle), au lieu de `y=t[y]`. De nombreux candidats ont adopté une solution récursive, l'itération d'ordre `k` appelant l'itération d'ordre `k-1`, sauf pour `k=0` ou `k=1` par exemple. Dans l'absolu, il faudrait également vérifier que le paramètre `k` est bien positif ; dans le cas contraire, on prend le risque d'une récursivité non controllée, qui va donc rapidement mal se terminer. L'absence de prise en compte de ce genre de situation n'a pas été sanctionnée.

Question 4 :
Cette question consiste à trouver le nombre de points fixes de la `k`-ième itérée d'un tableau. A partir des questions précédentes, cette question ne pose aucune difficulté.

Question 5 :

Cette question consiste à déterminer si une fonction représentée par un tableau admet un attracteur principal, autrement dit une valeur unique à laquelle conduisent toutes les itérations, quelque soit le point de départ. Cette question a posé plus de difficultés. Il fallait remarquer qu'une fonction admettant un attracteur principal ne peut pas admettre plusieurs points fixes. On trouve de très nombreuses variantes, allant des codes très simples qui se limitent à quelques lignes (tout en étant correctes) jusqu'à des codes très longs et compliqués. Il fallait que les candidats prennent vraiment le temps de réfléchir aux implications de la question avant de se lancer dans des codes complexes.

Question 6 :

Dans cette question, les candidats ont simplement à itérer jusqu'à atteindre le point fixe (qui est supposé exister) et à dénombrer les itérations nécessaires. L'énoncé suggère une solution récursive, qui correspond effectivement à la solution proposée dans la grande majorité des copies, bien qu'une boucle `while` puisse remplacer la récursion. Certaines approches sont à éviter, comme par exemple l'utilisation de la fonction `itere` écrite à la question 3, ou encore une recherche préliminaire du point fixe. Ces approches ne sont pas fausses, mais elles accroissent sensiblement le nombre d'opérations nécessaires sans pour autant simplifier l'écriture du code; elles sont donc naturellement moins récompensées que les autres.

Question 7 :

Dans cette question il faut calculer le temps de convergence maximal. La contrainte imposée d'un comportement linéaire en nombre d'opérations est bien entendu la difficulté principale. Un grand nombre de candidats se sont contentés d'appeler en boucle la fonction écrite dans la question précédente, puis de déterminer la valeur maximale du tableau des temps de convergence ainsi obtenu. Cette approche, de complexité quadratique, ne répond pas à la question. Il est à noter que certains candidats reconnaissent et assument la complexité quadratique de leur code, alors que d'autres prétendent avoir une solution de complexité linéaire.

A moins d'avoir eu recours à un mécanisme de memoisation lors de la question précédente, il faut éviter de faire appel à la fonction `temps_de_convergence`. Une solution consiste à maintenir un tableau intermédiaire qui mémorise les points déjà explorés lors des itérations antérieures ; ce tableau intermédiaire garde aussi en mémoire les temps de convergence déjà trouvés pour ces points. Dans ces conditions, le comportement linéaire est assuré par le fait qu'on ne calcule qu'une seule fois le temps de convergence d'un point donné.

Le barème de cette question a privilégié les copies qui présentaient une approche correcte, même si sa mise en œuvre n'était pas totalement valide. Le taux de réussite à cette question est plutôt faible.

Question 8 :

Cette question consiste à vérifier si une fonction est croissante en temps linéaire. Il suffit de vérifier que tout couple de valeurs consécutives est dans le bon ordre, en prenant garde aux débordements de tableaux. La comparaison de tout couple de deux valeurs quelconques en utilisant deux boucles imbriquées ne satisfaisait pas la contrainte de complexité. Afin de réduire le nombre d'opérations, il est possible d'interrompre par anticipation la boucle d'exploration dès que la condition de non-croissance est obtenue.

Question 9 :

Le point critique de cette question n'est pas la détermination du point fixe lui-même, mais bien la contrainte d'une complexité en temps logarithmique. De très nombreux candidats ont effectué un parcours exhaustif du tableau des valeurs de la fonction, conduisant à une complexité linéaire.

La fonction étant croissante, une dichotomie suffit pour déterminer la présence d'un éventuel point fixe Le principe de la dichotomie revient à diviser par 2 l'intervalle de recherche à chaque itération, ce qui garantit une complexité en temps logarithmique. Dans de nombreuses copies, l'idée de la dichotomie est exposée de façon assez claire, mais sa mise en œuvre n'est pas correcte. Il est très important, dans tous ces algorithmes de dichotomie, de clairement identifier par l'intermédiaire de variables dédiées les bornes minimale et maximale de l'intervalle en cours d'exploration, et non pas uniquement le point d'évaluation. La terminaison de la dichotomie est un autre point, plus subtil, que la question suivante se proposait de vérifier plus particulièrement.

Dans la mesure où la dichotomie est un schéma algorithmique que l'on retrouve dans de nombreuses situations, nous recommandons aux candidats de travailler ce point afin de garantir que ce schéma et son implémentation pratique soient bien compris.

La réussite à cette question est relativement mitigée compte tenu des remarques précédentes.

Question 10 :

Cette question découle en réalité directement du code écrit précédemment, dont il faut justifier le comportement logarithmique. Elle peut le cas échéant permettre de déceller une non-terminaison du code due à un problème d'arrondi. La plupart des candidats ayant réussi la question 9 ont pu justifier et donner une réponse satisfaisante à cette question.

Question 11 :

Cette question se traite par récurrence. Une très grande majorité des candidats ayant traité la question a obtenu la totalité des points. Un faible nombre n'a donné une preuve valable que pour le seul cas d'une relation d'ordre totale, preuve démarrant souvent par un raisonnement par l'absurde.

Question 12 :

Cette question ne pose pas de difficulté ; il faut néanmoins veiller à démontrer le critère de divisibilité dans les deux sens avant de conclure. Une grande majorité des candidats ayant traité la question a obtenu la totalité des points.

Question 13 :

Pour cette question il faut se servir des deux questions précédentes, et en particulier remarquer que 1 est le plus petit élément de l'ensemble au sens de la divisibilité. Il suffit donc de reprendre les itérations écrites dans les premières questions, en démarrant à partir de 1 et en interrompant la boucle dès qu'un point fixe est trouvé. Par application du résultat de la question 12, ce point fixe est précisément le pgcd de tous les points fixes.

Certains candidats ont calculé tous les points fixes et puis ont fait appel à une fonction de calcul du pgcd. Cette approche ne permettait pas de garantir la complexité attendue.

Dans la mesure où l'on arrive vers la fin de l'épreuve, cette question n'a tout simplement pas été abordée pour un nombre important de copies, soit par manque de temps, soit par manque d'idée.

Question 14 :

Cette question demande de justifier le comportement logarithmique du code écrit à la question 13, à savoir une itération simple en partant de 1. Il est clair que seuls les candidats ayant compris le principe de la question 13 étaient en mesure de répondre à cette dernière question. Concrètement, il faut comprendre que, selon la règle de divisibilité, soit on atteint le point fixe, soit l'itération multiplie la valeur courante par 2 au moins (puisque l'itération suit la règle de divisibilité), soit on multiplie par 0, et l'on termine alors à l'itération suivante. Ce critère de multiplication par 2 ou plus permet alors de rapidement conclure quant au comportement logarithmique de l'algorithme.