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

Rapport de MM. Samuel MIMRAM et Etienne LOZES, correcteurs.

1. Statistiques

Rappellons que cette épreuve n'est corrigée que pour les candidats admissibles. Cette épreuve est commune aux filières MP et PC, mais les copies étant indistinguables, les statistiques présentées intègrent ces deux filières.

$0 \leqslant \mathrm{N} < 4$	44	9 %
$4 \leqslant \mathrm{N} < 8$	157	31 %
$8 \leqslant \mathrm{N} < 12$	207	42 %
$12 \leqslant \mathrm{N} < 16$	80	16 %
$16 \leqslant \mathrm{N} \leqslant 20$	11	2 %
Total	499	100 %
Nombre de copies : 499		
Note moyenne : 8,77		
Écart-type : 3,58		
Note minimale : 0,3		
Note maximale : 19,3		

Les candidats ont majoritairement traité cette épreuve en répondant aux questions dans l'ordre, une minorité ayant omis les questions 9, 10 et 11 pour traiter directement les questions 12 et 13. Au final, 9% des candidats ont terminé cette épreuve, c'est-à-dire ont tenté de répondre aux 15 questions. Le choix du langage de programmation s'est porté sur Maple (88%), Python (10%), et plus rarement Mathematica, Pascal, ou C++.

2. Commentaires généraux

Le sujet portait sur la conception de fonctions de manipulation des permutations de $\{1, \ldots, n\} (n \in \mathbb{N})$ représentées par des tableaux d'entiers. Dans un premier temps, on étudiait leur structure de groupe (reconnaissance, composition, inversion), dans un second temps plusieurs grandeurs et propriétés propres à une permutation donnée (ordre, période d'un élément, transpositions, cycles), et dans un troisième temps on recherchait une algorithmique efficace pour les calculs de l'itérée k-ième d'une permutation et de son ordre.

Presque toutes les questions posées reposaient sur l'écriture d'une fonction dans un langage de programmation laissé au choix du candidat ; pour les questions 4 et 12, une

réponse mathématique était attendue, qui pouvait être complétée par un programme. Plusieurs primitives abstraites de constantes booléenes, de calcul de reste, et surtout de manipulation des tableaux étaient suggérées par l'énoncé. Ces dernières visaient à souligner les contraintes liées à la représentation mémoire des tableaux – notamment les problèmes d'allocation dynamique et de taille mémoire. Ces indications spécifiaient le type de manipulation attendu sur les tableaux. L'utilisation de primitives propres au langage de programmation choisi n'était pas explicitement interdite ; toutefois, le sujet proposant des primitives abstraites pour certaines opérations, il était attendu qu'elles soient utilisées, ou à défaut des primitives équivalentes dans le langage choisi. Les manipulations de tableaux plus avancées et spécifiques au langage de programmation choisi, notamment celles basées sur les correspondances entre listes et tableaux, n'ont pas été sanctionnées lorsqu'elles étaient maîtrisées, mais n'ont pas non plus été récompensées. Tenant compte du libre choix du langage de programmation à cette épreuve, il a été considéré qu'il convenait de valoriser la copie qui dépassait la seule connaissance d'un langage de programmation donné et concevait des solutions facilement transposables à d'autres langages de programmation. Dans cet esprit, les futurs candidats auront profit de s'entraîner à *utiliser les constructions élémentaires* (fonctions, boucles, tableaux...) de leur langage plutôt qu'à chercher à maîtriser les points les plus avancés.

En contrepartie, il est juste de constater qu'une majorité de candidats démontre une certaine familiarité avec le langage de programmation choisi. La minorité de candidats qui commet des erreurs grossières n'en apparaît que plus marginalisée. La correction syntaxique attendue est celle qui permet de reconstituer sans ambiguïté et avec peu d'efforts un code assimilable par un compilateur ou un interpréteur. La meilleure préparation à cette épreuve de programmation reste donc *une préparation pratique sur machine*. Plus concrètement, des erreurs moyennes telles que l'utilisation du `and` pour indiquer une composition séquentielle ont été considérées avec bienveillance cette année, de même que les erreurs minimes comme les oublis de fermeture de bloc d'instructions ou les confusions entre symbole d'affectation et d'égalité. Les erreurs graves, dépassant en réalité la simple erreur de syntaxe, comme le cas assez courant d'utilisation de boucles pour exprimer une conjonction

```
if for i from 1 to n t[i]=i then [..]
```

ont été, elles, sévèrement sanctionnées.

Des confusions de différents ordres entre le monde mathématique et celui informatique ont pu être relevées, notamment le recours aux indices, aux x_i de E_n (bien que l'énoncé indiquât de prendre $x_i = i$ et $E_n = \{1, \ldots, n\}$), voire aux exposants $\mathtt{t}^i$ pour `t` une variable de programme. Ces confusions n'ont en général pas été sanctionnées cette année, sauf dans le dernier cas lorsqu'elles ne consistaient pas en des identifiants de variables étendues, ou des annotations de code, mais à des indications de calcul erronées. Plus rare, mais révélateur d'une confusion répandue par Maple, l'utilisation hasardeuse du mot-clé *infinity* dans

```
for i from 1 to infinity do [..] od;
if i=infinity then [..]
```

qui suppose pour les ordinateurs une capacité de calcul qu'ils n'ont pas.

Les spécificités du langage de programmation choisi, ainsi que les hypothèses implicites (et parfois, de manière bienvenue explicites) sur les conventions adoptées sur ce langage ont pu faire débat. Il est souhaitable que les candidats s'interrogent sur les conventions induites par le choix d'un langage de programmation, qu'ils commentent et explicitent ces conventions, et qu'ils s'autorisent à adopter d'autres conventions, tant qu'elles sont explicitées et effectives sur l'ensemble du sujet.

Entiers machines. Le sujet ne précisait pas si les entiers machines étaient signés ou positifs. En l'absence de précision sur la convention adoptée par le candidat, un test de la forme $x = 0$ à la question 1, au lieu du test tout aussi simple mais plus robuste $x \leq 0$, a été très faiblement pénalisé.

Tableaux : allocation. L'allocation et le test de taille de tableaux étaient définis par l'énoncé. Les copies n'ayant pas utilisé l'allocation, ou ayant émis des hypothèses sur le contenu des tableaux fraîchement alloués ont été sanctionnées.

Tableaux : copie. La copie de tableau a été mise en œuvre par la grande majorité des candidats comme une simple affectation. Elle n'a jamais été sanctionnée. Les très rares copies ayant invoqué une primitive `x :=copy(t)` ou ayant écrit un code effectuant la copie ont été faiblement récompensées.

Tableaux : égalité. L'égalité de tableau n'était pas précisée par l'énoncé ; les langages les plus populaires chez les candidats présentent une égalité de tableau « extensionnelle », et non au sens d'« adresse mémoire », que l'on trouve néanmoins dans de nombreux autres langages de programmations. On a ainsi récompensé les candidats ayant témoigné d'une connaissance de cette subtilité. Le recours à une égalité extensionnelle facilitait en effet la conception du code, notamment à la question 5. En cas d'erreur, (le plus souvent, une erreur d'indice), les candidats ayant eu recours à l'égalité extensionnelle sans émettre de commentaires à ce sujet étaient plus pénalisés que ceux ayant fait la même erreur mais ayant reprogrammé correctement l'égalité extensionnelle.

Tableaux : indiçage. L'indiçage des tableaux à partir de 1 était imposé par l'énoncé ; la très grande majorité des copies a fort justement suivi cette convention. Les quelques copies ayant fait le choix d'un indiçage à partir de 0 ont presque toujours échoué aux questions 2 et 3 (composition et inversion).

Déroutements et récursivité. La plupart des langages choisis offrent des primitives de déroutement (`return, break, continue`, etc) qui permettaient parfois de clarifier le code ou de le rendre plus efficace, notamment aux questions 5, 6, et 7. De manière similaire, l'emploi de fonctions récursives simplifiait le code de la question 12. Il paraît

souhaitable que les candidats aient une meilleure connaissance de ces possibilités, car les rares copies les ayant exploitées ont généralement mieux répondu à ces questions.

Primitives propres au langage choisi. Comme indiqué précédemment, un certain nombre de primitives de langages permettaient parfois d'écrire un code plus concis, notamment le test booléen `k in t` en Python. Ce type de programmation n'a été sanctionné qu'en cas d'erreur, mais seuls les candidats capables de convaincre de leur maîtrise de la programmation devraient se l'autoriser. A titre d'exemple, certains candidats ayant opté pour Maple ont été pénalisés pour avoir converti un tableau en une séquence afin de déterminer si deux tableaux avaient le même ensemble de valeurs.

3. Commentaires par question

Un aperçu de la réussite par question est donné dans le tableau ci-dessous. Chaque question était notée sur 4 (avant pondération par le barème), la note maximale étant 5 (bonus). La première ligne indique le pourcentage de candidats ayant obtenu une note ≥ 4, la deuxième la moyenne, et la troisième le pourcentage de candidats ayant obtenu un zéro.

Question	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Réussite %	15	41	77	79	18	52	32	53	2	0,4	15	18	14	46	0,4
Moyenne	1,9	3,3	3,9	3,6	2,0	2,9	2,7	2,9	0,4	1,1	1,4	0,8	1,0	2,0	0,2
Zéro %	24	4	14	3	17	10	15	26	65	24	50	77	62	46	86

Question 1 : La solution juste la plus souvent rencontrée consistait à tester la surjectivité sur $\{1, \ldots, n\}$, comme suggéré par l'énoncé, l'autre solution correcte parfois rencontrée étant un test d'injectivité avec vérification que $\texttt{Im}(t) \subseteq \{1, \ldots, n\}$. De nombreux candidats n'ont testé qu'une seule condition dans la deuxième approche. Bien que cela fût inutile, certains candidats ont écrit, assez rarement avec succès, une procédure de tri visant à simplifier le test de surjectivité de la première approche. Un débordement de tableaux était possible pour certaines mises en œuvre de la première approche, comme dans le code érroné ci-dessous, ce qui coûtait au candidat un quart des points :

```
estPermutation := proc(t)
   local n,u ;
   n :=taille(t) ;
   u :=allouer(n) ;
   for i from 1 to n do u[i] :=0 od ;
   for i from 1 to n do u[t[i]] :=1 od ;
   for i from 1 to n do if u[i]=0 then return faux fi od ;
   return vrai
end ;
```

Question 2 : Bien que facile, cette question n'a pas rapporté le maximum de points à de nombreux candidats qui ont oublié d'allouer le tableau contenant le résultat, ou ont supposé que la taille des tableaux était contenue dans une variable globale. Quelques

copies ont effectué la composition $u \circ t$ au lieu de celle attendue.

Question 3 : Cette question a été généralement bien traitée, avec un nombre important de solutions efficaces en temps linéaire, qui ont été récompensées.

Question 4 : Cette question du domaine des mathématiques a généralement été bien traitée, les erreurs les plus courantes étant de donner une transposition comme exemple d'ordre 1 et la permutation « miroir » $i \mapsto n - i + 1$ comme exemple d'ordre n.

Question 5 : Cette question, comme la question 1, présentait de nombreux écueils, notamment sur le test d'arrêt de boucle, qui nécessitait de tester l'égalité de deux tableaux. Ce test a parfois été confondu avec la boucle d'itération principale, comme dans le code erroné ci-dessous :

```
ordre := proc(t)
   local i,k,u ;
   k :=1 ; u :=t ;
   for i from 1 to n do
     while t[i]<>i do k :=k+1 ; u :=composer(t,u) od
   od ;
   return k
end
```

Certains candidats ont choisi d'écrire une fonction auxiliaire pour la condition d'arrêt, et ont été récompensés pour leur code clair et modulaire.

Question 6 : Cette question a été généralement mieux traitée que la question 5, car la condition d'arrêt était plus simple. Les écueils communs aux questions 5 et 7 restaient cependant assez fréquents : le calcul des itérées de `t` dans une variable auxiliaire `u` était parfois mis en œuvre par `u :=composer(t,t)`, ou encore la valeur de retour différait de ± 1 de celle attendue soit sur toutes les entrées, soit seulement sur l'identité. Pour éviter ces erreurs, on ne saurait trop recommander aux candidats de tester mentalement leur code sur des exemples.

Question 7 : En plus des écueils communs aux questions 5 et 6, cette question présentait une difficulté supplémentaire sur la condition d'arrêt. Bien que la plupart des candidats aient choisi une condition très large (n, voire ordre(t) itérations), certains candidats ne faisaient que *periode(i)* -1 itérations, et ne parcouraient donc pas la totalité de l'orbite. Certains candidats ont proposé des solutions particulièrement efficaces - parcourant la suite des $t^k(i)$ sans faire appel à `composer`, et renvoyant le résultat aussitôt que celui-ci pouvait être déterminé - et ont dans ces conditions été récompensés pour leur code efficace.

Question 8 : Cette question admettait une solution simple qui consistait à comparer simplement à 2 le nombre de $i \in \{1, \ldots, n\}$ tels que $t(i) \neq i$. De nombreux candidats ont

trouvé cette solution efficace et ont été récompensés.

Question 9 : Cette question n'a que très rarement été traitée correctement. La plupart des candidats ont pensé qu'un cycle était une permutation dont exactement un élément serait de période non unitaire. Les quelques candidats qui ont noté qu'une telle définition n'admettrait aucun exemple de permutation cyclique ont été moins pénalisés que les autres. Les candidats ayant produit un code faux, mais ayant adopté la définition attendue de permutation cyclique, ont été encore moins pénalisés.

Question 10 : Il s'agissait sans doute de la question la plus difficile du sujet. La plupart des candidats ont donné une solution simple en temps quadratique, voire cubique. Ceux ayant relevé le problème ont été un peu moins pénalisés que ceux qui affirmaient avoir une solution en temps linéaire. Ceux ayant envisagé une solution plus complexe, mais néanmoins en temps non linéaire - par exemple en $O(\max(periode) \times n)$ - ont été encore un peu moins pénalisés. La solution attendue, consistant à parcourir chaque orbite deux fois (une pour calculer sa taille, l'autre pour inscrire cette taille dans le tableau des résultats) a été trouvée par de rares excellents candidats. Les candidats qui ont produit un code effectuant un parcours par orbite, mais faux, ont été moins pénalisés que ceux cités précédemment.

Question 11 : Le code à produire devait exploiter l'identité $t^k(i) = t^r(i)$ pour r le reste de k modulo $periode(i)$, et faire appel au tableau des périodes de la question 10. La question présentait de nombreux écueils sur le calcul de $t^r(i)$. La plupart des candidats ont cherché à calculer t^r par appel à `composer`, mais ont soit oublié de réinitialiser le tableau contenant t^r à chaque nouveau r, soit mal géré le cas $r = 0$, ce qui a conduit à un faible taux de réussite. Les copies ayant calculé $t^r(i)$ sans appeler `composer` ont été récompensées pour leur code efficace.

Question 12 : Pour de nombreux candidats, l'ordre d'une permutation ne peut excéder sa taille - ce qu'ils ont voulu parfois exploiter à la question 5. Un sixième des candidats a répondu correctement à cette question, qui relevait du domaine des mathématiques.

Question 13 : Cette question, bien que classique, présentait de nombreux écueils - soit sur l'utilisation de variables auxiliaires pour réaliser une double affectation, soit sur la condition d'arrêt, soit sur le résultat retourné. Les rares copies ayant utilisé la récursivité - ce que n'interdisait pas le sujet - ont presque toutes produit un code correct. Une minorité des copies ayant traité cette question a été pénalisée pour ne pas avoir suivi la consigne de l'énoncé qui imposait un calcul par l'algorithme d'Euclide.

Question 14 : Question très facile pour qui l'abordait. Dans leur précipitation, certains candidats ont néanmoins écrit un code approximatif, comme $\frac{a*b}{pgcd(a;b)}$ au lieu de $a * b/pgcd(a, b)$.

Question 15 : Question ouverte abordée par relativement peu de candidats, parmi lesquels environ la moitié n'a pas donné un code calculant l'ordre d'une permutation. La

solution correcte le plus souvent rencontrée, consistant à calculer le ppcm de *toutes* les périodes par accumulation, a été modérément récompensée. Un calcul basé sur une approche diviser pour régner, bien que coûtant lui aussi $n-1$ appels à `ppcm`, a été un peu mieux récompensé. La méthode la plus efficace proposée par certains candidats consistait à remplacer la boucle

`for i from 2 to n do o :=ppcm(o,per[i]) od` par

```
for i from 2 to n do if reste(o,per[i])<>0 then o :=ppcm(o,per[i]) fi od
```

Cette solution donnait le maximum de points, bien qu'on aurait pu encore réduire le nombre d'appels à `ppcm`, par exemple en triant le tableau des périodes par ordre décroissant.

Composition d'Informatique (2 heures), Filière MP (XC)

Rapport de M. Didier CASSEREAU, correcteur.

I. Bilan général

A titre de rappel, cette épreuve n'est corrigée que pour les candidats admissibles. Pour ma part, j'ai corrigé la filière MP.

Cette année le nombre total de candidats admissibles dans cette filière est de 280. La note moyenne est de 12,53 avec un écart-type de 3,45. 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 2,4/20 et la note maximale 18/20.

	Série 1		Série 2		Série 3		Série 4		**Synthèse**	
$0 \leq N < 4$	0	0,0%	1	1,4%	1	1,6%	0	0,0%	**2**	**0,7%**
$4 \leq N < 8$	15	16,1%	8	11,4%	4	6,3%	6	11,1%	**33**	**11,8%**
$8 \leq N < 12$	23	24,7%	24	34,3%	19	30,2%	15	27,8%	**81**	**28,9%**
$12 \leq N < 16$	38	40,9%	26	37,1%	23	36,5%	25	46,3%	**112**	**40,0%**
$20 \leq N \leq 20$	17	18,3%	11	15,7%	16	25,4%	8	14,8%	**52**	**18,6%**
Total	93	100,0%	70	100,0%	63	100,0%	54	100,0%	**280**	**100,0%**
Epreuve complète*	46	49,5%	34	48,6%	32	50,8%	26	48,1%	**138**	**49,3%**

* *Epreuve complète* signifie ici que le candidat a abordé toutes les questions de l'énoncé et obtenu une note non nulle à chacune des 15 questions.

	Série 1	Série 2	Série 3	Série 4	**Synthèse**
Nombre de copies	93	70	63	54	**280**
Note minimale	4,1	2,4	3,7	5,3	**2,4**
Note maximale	18,0	18,0	17,7	18,0	**18**
Note moyenne	12,48	12,27	12,98	12,42	**12,53**
Ecart-type	3,51	3,64	3,31	3,28	**3,45**

Le langage de programmation choisi par les candidats est largement dominé par Maple (encore mal orthographié par bon nombre de candidats) avec 63,2% des copies, suivi ensuite par Caml (24,6% des copies) et C/C++ (6,1% des copies). On trouve enfin quelques copies en Java (2,9%), Python (1,8%), Pascal (0,7%) et autres (0,7%, incluant par exemple Mathematica). Le tableau ci-dessous illustre cette répartition des langages choisis par les candidats.

Maple	Caml	C/C++	Java	Python	Pascal	Autres	**Total**
177	69	17	8	5	2	2	**280**
63,2%	24,6%	6,1%	2,9%	1,8%	0,7%	0,7%	**100,0%**

II. Commentaires

Cette année le sujet portait sur les permutations, avec 3 phases :

- la première partie abordait les notions fondamentales liées aux permutations, aboutissant au calcul de l'ordre d'une permutation,
- la deuxième partie consistait à manipuler plus précisément les permutations afin d'en dégager des caractéristiques importantes telles que transposition, cycle,...
- la troisième et dernière partie était un peu plus complexe et visait à minimiser le nombre d'opérations faites par le programme dans les opérations de manipulation des permutations.

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 en C par exemple). Cette directive a été largement respectée par la grande majorité des candidats.

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.

J'invite en particulier les candidats composant dans des langages tels que Mathematica à être très vigilants sur ce point.

L'évaluation d'un code informatique repose sur plusieurs éléments à mon avis essentiels, parmi lesquels

- évidemment le code doit être juste et donner le résultat correct,
- la clarté et la lisibilité du code sont également des éléments essentiels de l'évaluation : l'algorithme mis en œuvre doit être simple et clair, et un soin particulier doit être apporté dans la manière de présenter le code (indentation des boucles et tests, passages à la ligne,...),
- les commentaires explicatifs ne sont pas strictement indispensables, ils peuvent néanmoins aider à comprendre la démarche mise en œuvre, en particulier lorsque la méthode utilisée n'est pas simple ou ne fonctionne pas,
- l'efficacité intervient également dans l'évaluation du code, même si cela n'apparaît pas explicitement dans l'énoncé ; certains candidats peuvent ainsi se voir attribuer un bonus efficacité pour le code qu'ils ont produit, ce bonus devenant l'essentiel de la note si l'aspect efficacité du code est explicite dans la question.

Quelques remarques générales à la lecture des codes :

- on trouve énormément d'instructions totalement inutiles, dans le genre `x=x` ou `a=a*1`, ces instructions ne servent à rien sinon à obscurcir le code, il serait bien de les éviter,
- on ne peut pas mettre une boucle `for` ou `while` dans la condition d'un test ; par contre on peut écrire une fonction séparée qui implémente la boucle et retourne un résultat du type vrai/faux, il suffit alors de tester la valeur retournée par cette fonction,
- on observe souvent un usage abusif de la récursivité ; dans certains cas il est très pratique de pouvoir faire appel à cette spécificité de certains langages de programmation, il faut cependant bien avoir conscience que cela peut rendre le code plus complexe à appréhender, et que c'est plus consommateur en ressources machine qu'une simple boucle `for` ou `while` ; de manière générale je conseille de restreindre l'utilisation de code récursif aux situations pour lesquelles cela apporte vraiment quelque chose en terme de simplicité algorithmique,
- 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 totalement tordues avec un code très difficile à lire et à comprendre ; clairement une telle situation doit être évitée,
- partant de l'exemple concret de la fonction `periodes(t)` qui calcule le tableau des périodes de la permutation `t`, il faut bien comprendre qu'à chaque fois que le programme contient l'instruction `periodes(t)`, cela correspond concrètement à un appel de fonction, et donc à l'exécution de toute une série d'instructions plus ou moins coûteuses en nombre d'opérations ; ainsi une fonction qui comporte une multitude d'instructions de la forme `periodes(t)` sera très peu efficace, ce genre de situation est donc à éviter,
- certaines notations n'ont pas de sens dans un langage informatique, telles que par exemple `t'`, $\texttt{t}^{-1}$, ou encore les accents dans les noms de variables ou fonctions,
- je mets en garde les candidats contre le calcul de `n!` qui est souvent beaucoup plus problématique qu'on ne l'imagine.
- il faut faire attention à ne pas manipuler des variables qui ne sont pas initialisées,
- 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 ; rien ne doit être approximatif dans le domaine de l'informatique, il faut au contraire veiller à être très précis,
- je suis assez surpris de constater que certains candidats n'utilisent que des boucles `while`, alors que dans certaines circonstances la boucle `for` est plus intuitive et plus simple.

III. Commentaires détaillés

Pour chaque question, un tableau récapitule les taux de réussite avec les conventions suivantes :

- 0 signifie aucun point pour la question (question non traitée ou abordée mais totalement fausse, ce second cas de figure étant rare),
- $< 0,5$ signifie moins de la moitié des points de la question,
- $\geq 0,5$ signifie plus de la moitié des points de la question,
- 1 signifie la totalité des points de la question.

Question 1 :

	0		<0,5		≥0,5		1		Total	
Question 1	9	3,2%	100	35,7%	160	57,1%	11	3,9%	280	100,0%

Cette question consiste à vérifier si un tableau de valeurs entières correspond à une permutation. Cela suppose de vérifier que toutes les valeurs du tableau sont comprises entre 1 et `n` (dimension du tableau) d'une part, que les valeurs sont 2 à 2 distinctes d'autre part.

Une méthode linéaire consiste à utiliser un tableau intermédiaire qui enregistre chaque valeur déjà rencontrée précédemment dans la boucle ; une erreur fréquente consiste alors à oublier de vérifier d'abord que les `t[i]` sont bien entre 1 et `n` ; dans le cas contraire cela provoque un débordement de tableau.

Beaucoup de candidats ont utilisé une double boucle, avec une logique plus ou moins bien maîtrisée, ce qui dans certains cas conduit à des codes assez farfelus.

Question 2 :

	0		<0,5		≥0,5		1		Total	
Question 2	4	1,4%	11	3,9%	258	92,1%	7	2,5%	280	100,0%

Cette question consiste à calculer la composée de deux permutations ; elle n'a pas vraiment posé de grosses difficultés. Il peut cependant être utile de vérifier que les deux permutations transmises à la fonction ont des tailles compatibles ; un bonus a été donné aux candidats qui l'ont vérifié, ou qui l'ont au moins mentionné à titre de réserve.

Question 3 :

	0		<0,5		≥0,5		1		Total	
Question 3	10	3,6%	0	0,0%	43	15,4%	227	81,1%	280	100,0%

Cette question consiste à calculer l'inverse d'une permutation. Là encore cette question n'est pas compliquée et a été bien traitée par une grande majorité de candidats.

A noter que ce calcul peut se faire avec une seule boucle ; il n'est absolument pas utile de faire deux boucles imbriquées, la seconde boucle cherchant une valeur `i` donnée dans le tableau.

Certains candidats ont fait la confusion entre l'inverse et le miroir.

Question 4 :

	0		<0,5		≥0,5		1		Total	
Question 4	1	0,4%	0	0,0%	21	7,5%	258	92,1%	280	100,0%

Cette question est extrèmement simple puisqu'il suffisait de donner un exemple de permutation d'ordre 1 (l'identité) et un exemple de permutation d'ordre `n` (un `n`-cycle par exemple).

Je pense qu'il est aisé de se convaincre que `[ 2 1 3 4 5 ]` par exemple n'est pas d'ordre 1, comme un certain nombre de candidats semble le croire.

Question 5 :

	0		<0,5		≥0,5		1		Total	
Question 5	4	1,4%	88	31,4%	164	58,6%	24	8,6%	280	100,0%

Cette question consiste à calculer l'ordre d'une permutation à partir de la fonction `composer()`. Le principe consiste à itérer l'appel à `composer()` jusqu'à ce que l'on trouve la permutation identité. Il faut pour cela utiliser une permutation `taux`, correctement initialisée, et itérer l'instruction `taux=composer(t,taux)`.

Une erreur assez fréquente est d'itérer l'instruction `t=composer(t,t)`, ce qui bien entendu nous amène à calculer en réalité $\texttt{t}^2$, $\texttt{t}^4$, $\texttt{t}^8$,... Dans certains cas, l'utilisation de la récursivité a contribué à cette erreur.

Une autre erreur assez fréquente consiste à assurer la condition d'identité pour les différents indices *successivement*, et non *simultanément*.

Dans cet exercice, j'ai vu énormément de tests avec une boucle `for` ou `while` dans la condition du test, ce qui ne peut pas fonctionner. Il est ici conseillé d'écrire une fonction annexe qui compare `taux` à l'identité, cela facilite beaucoup les choses.

Le principe du code est ici d'itérer ce qui suppose que la boucle se termine vraiment. C'est ce cas, cela peut se démontrer mathématiquement pour les permutations ; un bonus a été donné aux candidats qui le faisaient remarquer, dans la mesure où cela implique qu'ils se sont au moins posé la question.

Question 6 :

	0		<0,5		≥0,5		1		Total	
Question 6	2	0,7%	31	11,1%	123	43,9%	124	44,3%	280	100,0%

Cette question consiste à calculer la période d'un élément `i` de la permutation. Elle peut se comprendre comme une variante de la question précédente, à la restriction près que l'utilisation de la fonction `composer()` nous fait calculer toutes les valeurs de la nouvelle permutation, alors qu'on n'a besoin que d'une valeur d'indice particulier. Cela fait donc des opérations inutiles qu'on peut éviter.

Une autre maladresse, mais qui n'est pas une erreur, a consisté (pour cette question et la question précédente) à utiliser une fonction annexe qui calcule la puissance `k` d'une permutation, puis à appeler cette fonction en boucle avec une valeur de `k` qui s'incrémente d'une unité à chaque nouvelle boucle. L'inconvénient majeur de cette méthode est que l'on reprend tout le calcul depuis le départ pour toute nouvelle valeur de `k`, ce qui fait rapidement un nombre d'opérations important.

Question 7 :

	0		<0,5		≥0,5		1		Total	
Question 7	7	2,5%	42	15,0%	124	44,3%	107	38,2%	280	100,0%

Cette question consiste à déterminer si un indice `j` donné est dans la même orbite qu'un indice `i`. Cette question est assez similaire à la précédente, les candidats ayant souvent implémenté les mêmes boucles dans les deux cas.

Un piège à éviter : l'indice `j` peut ne pas être dans l'orbite de `i`, il faut donc veiller à ne pas faire une boucle infinie. Globalement ce piège a plutôt été bien vu et esquivé par une grande majorité de candidats.

Question 8 :

	0		<0,5		≥0,5		1		Total	
Question 8	14	5,0%	39	13,9%	42	15,0%	185	66,1%	280	100,0%

Dans cette question on cherche à déterminer si une permutation est une transposition, à savoir qu'elle n'a que des singletons hormis une paire unique.

Cette question n'est pas très compliquée et a été généralement bien traitée, hormis le fait que certains candidats sortent des algorithmes très compliqués alors que le critère à vérifier est finalement relativement simple.

Les candidats doivent lire attentivement l'énoncé, en particulier l'identité n'est pas une transposition.

Question 9 :

	0		<0,5		≥0,5		1		Total	
Question 9	44	15,7%	82	29,3%	64	22,9%	90	32,1%	280	100,0%

Dans cette question on cherche à déterminer si une permutation est un cycle, à savoir qu'elle n'a que des singletons hormis une orbite unique de longueur supérieure à 1 (là encore il faut lire attentivement l'énoncé, l'identité n'est pas un cycle selon la définition qui en est donnée).

Une erreur très fréquente a consisté à calculer le nombre de périodes supérieures à 1, et à vérifier ensuite que ce nombre était égal à 1. En réalité ce nombre ne peut jamais être 1 puisque l'on explore tous les indices, en particulier toutes les valeurs qui sont dans la seule orbite de taille supérieure à 1. Par contre ces orbites doivent être toutes identiques, et c'est ce qu'il fallait vérifier.

Question 10 :

	0		<0,5		≥0,5		1		Total	
Question 10	63	22,5%	125	44,6%	26	9,3%	66	23,6%	280	100,0%

Cette question est sans doute une des plus délicates, dans la mesure où l'on veut calculer le tableau des périodes, mais en imposant un coût linéaire.

Une solution très souvent proposée consiste en une boucle dans laquelle on appelle la fonction `periode()` de la question 6 pour chaque indice. Cette solution est correcte, mais ne satisfait pas la contrainte de linéarité.

On peut utiliser la propriété que toutes les valeurs d'une même orbite ont la même période. On boucle sur le tableau des périodes ; pour chaque nouvelle orbite non encore rencontrée, on calcule sa période, puis on propage cette période à toute l'orbite. Il faut bien entendu veiller à ne pas traiter plusieurs fois la même orbite pour conserver le coût linéaire.

Question 11 :

	0		<0,5		≥0,5		1		Total	
Question 11	27	9,6%	65	23,2%	80	28,6%	108	38,6%	280	100,0%

Dans cette question, on souhaite itérer `k` fois une permutation de manière efficace. La clé était fournie dans l'énoncé qui proposait d'utiliser le reste de la division entière de `k` par la période de chaque indice.

L'efficacité est ainsi garantie par 1) l'appel de la fonction `periodes()` efficace précédente, 2) le calcul du reste de la division de `k` par chaque valeur du tableau des périodes, et 3) l'itération proprement dite.

Par souci d'efficacité qui est ici une priorité, il faut éviter de faire appel à la fonction `composer` qui nous fait faire des opérations inutiles.

Toujours par souci d'efficacité, il ne faut pas faire appel plusieurs fois à `periodes()`, ni calculer en boucle le reste de la division entière de `k` par les différentes périodes.

Question 12 :

	0		<0,5		≥0,5		1		Total	
Question 12	70	25,0%	0	0,0%	25	8,9%	185	66,1%	280	100,0%

Cette question consiste à fournir un exemple de permutation dont l'ordre excède la taille. La dernière phrase du paragraphe précédent donnait un indice très important qui aidait beaucoup dans la logique de la réflexion.

On peut ainsi chercher en direction d'un 2-cycle joint à un 3-cycle, ce qui nous donne une permutation de taille 5 et d'ordre ppcm(2,3)=6 : `[ 2 1 5 3 4 ]` par exemple.

La question est posée sous une forme assez affirmative quant à l'existence d'une telle permutation (*Donner un exemple...*) ; malgré cela certains candidats n'hésitent pas à écrire qu'un tel exemple ne peut pas exister.

Question 13 :

	0		<0,5		≥0,5		1		Total	
Question 13	23	8,2%	41	14,6%	35	12,5%	181	64,6%	280	100,0%

Un grand classique, mais qui donne souvent lieu à... n'importe quoi ! On cherche à calculer le plus grand diviseur commun entre 2 nombres entiers par l'algorithme d'Euclide.

Ce calcul est très classique mais aussi souvent faux (une itération en trop, la condition d'arrêt qui est mal gérée, l'itération récursive écrite n'importe comment...). On y trouve un peu de tout !

Parmi les choses qu'il faudrait bien vérifier : 1) vérifier que les valeurs ne sont pas nulles (sinon c'est direct la division par 0), 2) gérer proprement les cas `a>b` et `a<b`, et 3) ne pas écrire n'importe comment un appel récursif !

Noter que l'algorithme d'Euclide peut s'écrire récursivement, mais également classiquement avec une simple boucle `while`.

Question 14 :

	0		<0,5		≥0,5		1		Total	
Question 14	22	7,9%	0	0,0%	0	0,0%	258	92,1%	280	100,0%

Cette fonction consiste à calculer le plus petit commun multiple entre 2 nombres entiers. Normalement cela devrait être évident, compte tenu de la relation classique `ppcm(a,b)*pgcd(a,b)=a*b`. Apparemment certains candidats ne connaissent pas cette égalité.

Question 15 :

	0		<0,5		≥0,5		1		Total	
Question 15	58	20,7%	9	3,2%	50	17,9%	163	58,2%	280	100,0%

Cette question consiste à calculer de manière efficace l'ordre d'une permutation. A partir des éléments précédents, la question n'est pas vraiment compliquée, un premier appel à la fonction `periodes()` efficace (ne pas l'utiliser plusieurs fois), suivi d'une boucle de calcul appelant `ppcm` entre les différentes périodes élémentaires. Un petit bonus en terme d'efficacité consiste à ne pas prendre en compte plusieurs fois la même période, et d'ignorer les singletons.