Composition d'Informatique (2 heures), Filière MP

1 Bilan général

À titre de rappel, cette épreuve n'est corrigée que pour les candidats admissibles. Le présent rapport ne concerne que la filière MP. Cette année le nombre total de candidats admissibles dans cette filière est de 557. La note moyenne est de 13,0 avec un écart-type de 3,1. La note minimale est de 2,6/20 et la note maximale 20/20. Aucune copie n'a obtenu une note éliminatoire. Au total, 59 candidats (10, 6%) ont traité le sujet dans son intégralité, c'est-à-dire ont obtenu une note strictement positive à chacune des 16 questions. La répartition des notes est résumée dans le tableau qui suit.

	$[0;4[$	$[4;8[$	$[8;12[$	$[12;16[$	$[16;20]$
Effectif	1 (0,2%)	28 (5,0%)	181 (32,5%)	248 (44,5%)	99 (17,8%)

2 Commentaires

Le sujet portait cette année sur le calcul de l'intersection de deux ensembles de points à coordonnées entières. La première partie abordait l'implémentation d'une solution naïve en Python ; la seconde partie demandait d'écrire des requêtes SQL liées à ce calcul d'intersection ; les trois dernières parties introduisaient une structure de données pour les points (codage de Lebesgue) et les ensembles de points (AQL) pour aboutir à un calcul plus efficace de l'intersection de deux ensembles de points.

Quelques remarques viennent à la première lecture des copies ; l'une des spécificités du sujet était d'imposer aux candidats l'utilisation d'un sous-ensemble des opérations sur les listes du langage Python, en particulier pour des raisons d'efficacité et de calcul de complexité. Le non respect de cette consigne a été à l'origine de beaucoup d'erreurs. Il est rappelé l'importance de lire le sujet dans son intégralité avant de traiter les questions. On peut citer entre autres l'utilisation non autorisée :

- des tranches de liste (syntaxe de type `l[i:j]`), ce qui par ailleurs peut changer la complexité d'une solution – au besoin, il était plus judicieux d'utiliser une fonction auxiliaire pour comparer des portions de listes plutôt que de les construire ;
- de l'ajout en tête de liste (là aussi la question de la complexité se pose) ;
- de la génération de liste utilisant une syntaxe de type `[f(x) for x in ... if ...]`, par ailleurs parfois mal maitrisée par les candidats.

Attention aux algorithmes récursifs : s'ils ne sont pas terminaux, ou fabriquent des tranches de listes (ce qui n'était de toute façon pas autorisé), la complexité est bien plus grande que prévue a priori. Et ils ne sont pas forcément plus facilement justes.

Il est en général inutile de traiter à part le cas des listes vides, les boucles fonctionnent correctement sur les listes vides.

On rappelle qu'il est possible d'itérer sur les élements d'une liste (syntaxe de type `for x in l: ...`) ; quand il n'est pas nécessaire d'utiliser un indice, cela permet d'écrire un code plus court, tout aussi lisible, et évite des erreurs de décalage d'indice.

De façon plus anecdotique, on voit encore souvent dans les copies des tests de type `if test == True: ...`, où `test` est une variable booléenne ; bien que cela ne constitue pas une faute en soi, c'est assez peu élégant et devrait être évité.

Enfin, un certain nombre de candidats ont négligé voire complètement fait l'impasse sur les questions relatives à SQL ; cela est déconseillé et a été relativement pénalisant. Parmi les erreurs fréquemment rencontrées pour la partie concernant SQL, un certain nombre de requêtes proposées présentaient des

ambiguïtés. Penser à nommer les tables ou sous-requêtes (en utilisant le mot-clé `AS`) quand cela est nécessaire.

3 Commentaires détaillés

Pour chaque question, on présente la répartition des résultats suivant les quatre catégories suivantes :
— $N = 0$ si la question est non traitée ou la réponse est complètement fausse,
— $N \in]0; 0,5[$ si la question est partiellement traitée ou la réponse comporte de nombreuses erreurs,
— $N \in [0,5; 1[$ si la question est traitée correctement avec quelques erreurs,
— $N = 1$ si la question est traitée correctement.

Question 1

	0	$]0; 0,5[$	$[0,5; 1[$	1
Question 1	28 (5,0%)	2 (0,4%)	30 (5,4%)	497 (89,2%)

Pour cette question, on ne pouvait se contenter d'utiliser l'opérateur `in`, qui n'était pas autorisé par le sujet. Par ailleurs, on préfère les solutions dont la complexité moyenne est de $l(q)/2$, où $l(q)$ est la longueur de la liste en entrée. Pour les candidats ayant utilisé une boucle `while`, ne pas oublier d'incrémenter l'indice dans la boucle.

Question 2

	0	$]0; 0,5[$	$[0,5; 1[$	1
Question 2	9 (1,6%)	1 (0,2%)	24 (4,3%)	523 (93,9%)

Certains candidats ont oublié de retourner la liste construite. Par ailleurs, l'énoncé indiquait que l'ensemble de points était représenté *« par une liste de points sans répétition »*, il n'était donc pas nécessaire de chercher à retirer les doublons.

Question 3

	0	$]0; 0,5[$	$[0,5; 1[$	1
Question 3	8 (1,4%)	0 (0,0%)	28 (5,0%)	521 (93,5%)

Il est demandé de justifier un minimum la complexité proposée.

Question 4

	0	$]0; 0,5[$	$[0,5; 1[$	1
Question 4	34 (6,1%)	3 (0,5%)	59 (10,6%)	461 (82,8%)

Attention à la syntaxe, il faut écrire `POINTS.x`, et non `x.POINTS`. Dans la clause `WHERE`, il faut utiliser l'opérateur `IN` et non `=` lorque l'on souhaite filter les entrées membres d'un ensemble. Certaines copies présentent les requêtes avec passages à la ligne et indentation judicieusement choisis, ce qui rend la solution plus lisible.

Question 5

	0	$]0; 0,5[$	$[0,5; 1[$	1
Question 5	247 (44,3%)	10 (1,8%)	70 (12,6%)	230 (41,3%)

Dans un certain nombre de cas, il manquait une jointure, ce qui revenait à écrire une requête (absurde) cherchant les enregistrements de la table `MEMBRE` pour lesquels `MEMBRE.idensemble` était à la fois égal à i et à j.

Une autre erreur couramment rencontrée a été de calculer l'union plutôt que l'intersection. La stratégie qui consiste à tester que `MEMBRE.idensemble` soit égal à i ou à j pouvait fonctionner, à condition de regrouper par `MEMBRE.idpoint` et de compter le nombre d'éléments retournés.

Question 6

	0	$]0;0,5[$	$[0,5;1[$	1
Question 6	173 (31,1%)	5 (0,9%)	75 (13,5%)	304 (54,6%)

Il était possible de réutiliser la requête de la question 4, en l'imbriquant dans la requête pour cette question. Attention là aussi à bien faire les jointures nécessaires, certaines solutions proposées ne renvoient (au mieux) que l'identifiant du point de coordonnées (a, b).

Question 7

	0	$]0;0,5[$	$[0,5;1[$	1
Question 7	40 (7,2%)	6 (1,1%)	19 (3,4%)	492 (88,3%)

Il est conseillé aux candidats de détailler un minimum le calcul de la solution proposée, de façon à ce qu'en cas de réponse incorrecte on puisse évaluer la gravité de l'erreur. À défaut, le correcteur est obligé de sanctionner lourdement. Parmi les erreurs récurrentes, certains candidats ont inversé le rôle de l'abscisse et de l'ordonnée.

Question 8

	0	$]0;0,5[$	$[0,5;1[$	1
Question 8	33 (5,9%)	57 (10,2%)	222 (39,9%)	245 (44,0%)

Un certain nombre de candidats ont présenté des solutions très « verbeuses » pour la conversion en base 4 des couples de bits ; il suffisait simplement d'écrire `2*bits(p[0], k) + bits(p[1], k)`. Par ailleurs, il était possible de commencer à traiter les bits de poids faible ou les bits de poids fort. On rappelle que la syntaxe en Python pour itérer à rebours de `n-1` à `0` (inclus) est : `for k in range(n-1, -1, -1): ...`. Si cette syntaxe (peu intuitive) est mal maitrisée, il est préférable d'écrire un code équivalent utilisant un parcours croissant des indices. Dans tous les cas, il fallait faire attention au sens de lecture pour la construction de la solution. Enfin, ne pas oublier de renvoyer le résultat à la fin de la fonction `code`.

Question 9

	0	$]0;0,5[$	$[0,5;1[$	1
Question 9	2 (0,4%)	1 (0,2%)	21 (3,8%)	533 (95,7%)

Cette question n'a pas posé de difficulté particulière ; elle a par ailleurs permis d'éviter de sur-sanctionner un candidat ayant compris la définition de comparaison à l'envers.

Question 10

	0	$]0;0,5[$	$[0,5;1[$	1
Question 10	47 (8,4%)	15 (2,7%)	73 (13,1%)	422 (75,8%)

Cette question demandait d'implémenter une relation d'ordre lexicographique. Il fallait naturellement commencer par comparer les bits de poids fort, c'est-à-dire faire une lecture de gauche à droite. La mise en place de la boucle `while` a posé un certain nombre de problèmes : test d'indice maladroit voire pas de test de dépassement, pas d'incrémentation de l'indice dans la boucle. Enfin, le cas d'égalité ne peut se détecter qu'après un parcours complet des deux codages de Lebesgue, ce n'est donc pas correct de renvoyer un `0` prématurément, c'est-à-dire à l'intérieur de la boucle.

Question 11

	0	$]0;0,5[$	$[0,5;1[$	1
Question 11	23 (4,1%)	0 (0,0%)	38 (6,8%)	496 (89,0%)

L'énoncé demandait (à tort) un codage de Lebesgue *« compacté »*, certains candidats ont écrit « 3 », au lieu de « 34 » qui aurait été conforme à la notation introduite à la question suivante. Bien entendu, cela n'a pas été sanctionné.

Question 12

	0	$]0;0,5[$	$[0,5;1[$	1
Question 12	38 (6,8%)	1 (0,2%)	58 (10,4%)	460 (82,6%)

Cette question n'a pas posé de difficulté particulière ; à l'instar de la question 11, elle était prioritairement destinée à inviter les candidats à travailler sur un exemple afin d'assimiler les concepts de codage de Lebesgue et d'AQL.

Question 13

	0	$]0;0,5[$	$[0,5;1[$	1
Question 13	33 (5,9%)	73 (13,1%)	269 (48,3%)	182 (32,7%)

Le sujet précisait que la liste `q` en entrée *« représenta[it] le codage de Lebesgue compacté d'un quadrant »* ; il était donc possible de ne pas tester toute la fin de la liste. Il était demandé de créer une *nouvelle* liste pour éviter de modifier la liste passée en paramètre ; à noter que la syntaxe `r = q` ne crée pas de nouvelle liste mais copie simplement la référence de la liste `q` dans une nouvelle variable, il faut écrire `r = list(q)` – syntaxe rappelée en début d'énoncé.

Question 14

	0	$]0;0,5[$	$[0,5;1[$	1
Question 14	290 (52,2%)	83 (14,9%)	128 (23,0%)	55 (9,9%)

Cette question complexe comportait plusieurs difficultés, la première consistait à identifier qu'un groupe de 4 AQL consécutifs formait un quadrant. La solution la plus élégante consistait probablement à tester l'égalité des préfixes des premier et quatrième AQL, ce que l'on pouvait faire en s'aidant de `ksuffixe`, cette fonction ne modifiant pas la liste passée en paramètre. On pouvait aussi écrire une fonction de comparaison *ad hoc*.

De manière générale, il fallait faire attention à la complexité… l'objectif de cette partie est de faire mieux qu'une complexité quadratique.

Attention si l'on utilise une boucle `while i < len(s)-3`, il faut penser à ajouter tout de même les 3 derniers éléments.

On rappelle que « supprimer » des éléments dans une liste, cela coûte très cher ! Il vaut mieux remplir une autre liste à mesure. De même, supprimer les doublons après modification peut coûter très cher selon la méthode utilisée.

Il ne faut pas se contenter de supprimer les doublons après toutes les itérations sur `k` : il est nécessaire de les enlever à mesure pour que les ensembles de points se retrouvent voisins dans la liste, pour pouvoir continuer à compacter.

Question 15

	0	$]0;0,5[$	$[0,5;1[$	1
Question 15	204 (36,6%)	12 (2,2%)	54 (9,7%)	287 (51,5%)

La structure de `compare_ccodes` est similaire à celle de `compare_pcodes` (question 10). Les difficultés rencontrées sont elles aussi comparables ; par ailleurs, certains candidats ont cherché à réutiliser `compare_pcodes`, le plus souvent de façon maladroite.

Question 16

	0	$]0;0,5[$	$[0,5;1[$	1
Question 16	355 (63,7%)	25 (4,5%)	76 (13,6%)	101 (18,1%)

La structure naturelle de la fonction `intersection2` est proche de celle d'un tri fusion, cependant contrairement à ce dernier c'est une intersection que l'on calcule, il ne faut donc pas terminer par inclure la dernière liste non vide dans le résultat final. Certains candidats ont d'ailleurs calculé l'union plutôt que l'intersection.

Attention de ne pas appeler inutilement `compare_ccodes` plusieurs fois.

Certains candidats n'ont pas exploité le fait que les listes en entrée étaient triées, la complexité de la solution proposée dans ce cas est $l(p) \times l(q)$, et non $l(p) + l(q)$.

À noter que l'appel à la fonction `compacte` était inutile ici : en effet, aucun « nouveau » quadrant ne peut apparaitre dans l'intersection.

1 Bilan général

À titre de rappel, cette épreuve n'est corrigée que pour les candidats admissibles. Le présent rapport ne concerne que la filière PC.

Cette année, le nombre total de candidats admissibles dans cette filière est 537. La note moyenne est 10.62 avec un écart-type de 2.76. 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 2.10 et la note maximale 19.60.

	Série 1		Série 2		Série 3		Série 4		Synthèse	
$0 \leq N \leq 4$	5	3%	2	1%	1	0%	1	0%	9	1%
$4 < N \leq 8$	28	20%	16	11%	20	14%	20	15%	84	15%
$8 < N \leq 8$	59	44%	78	57%	76	55%	63	48%	276	51%
$12 < N \leq 16$	33	24%	38	27%	37	27%	42	32%	150	27%
$16 < N \leq 20$	9	6%	2	1%	2	1%	5	3%	18	3%

	Série 1	Série 2	Série 3	Série 4	Synthèse
Nombre de copies	134	136	136	131	537
Notes minimales	3.00	3.30	2.10	3.30	2.10
Notes maximales	19.60	16.90	16.60	17.80	19.60
Notes moyennes	10.55	10.59	10.53	10.82	10.62
Écart-type	3.26	2.44	2.47	2.79	2.76

Le langage de programmation PYTHON a été utilisé pour la totalité des copies.

2 Commentaires

L'épreuve de cette année portait sur le calcul de l'intersection entre deux ensembles de points de D_n^2 où D_n représente l'ensemble des entiers entre 0 et $2^n - 1$. On étudiait trois méthodes : un algorithme naïf, une solution s'appuyant sur des bases de données relationnelles et une dernière solution fondée sur une structure de données dite d'AQL.
D'une manière générale, les aspects suivants sont pris en compte (par ordre d'importance) dans la correction :

1. la correction de l'*algorithme* proposé vis-à-vis de la spécification fournie ;
2. l'efficacité de la solution ;
3. la lisibilité du code (on favorise une solution simple devant une solution compliquée équivalente, on apprécie le code bien indenté et structuré, ...) ;
4. la correction des détails d'*implémentation* ;
5. les explications accompagnant le code sous la forme de commentaires.

Les correcteurs tiennent à faire remarquer les points de forme suivants :

— En Python, l'indentation est significative : ne pas indenter de façon suffisamment explicite peut donc rendre la réponse du candidat ambigüe.
— Il faut éviter d'introduire une fonction auxiliaire pour ne finalement pas l'utiliser dans la réponse finale. Le correcteur perd du temps bêtement à comprendre ce code inutile. Rayer proprement une partie inutile de la réponse est préférable dans cette situation.
— Pour améliorer la présentation, il est déconseillé de commencer à écrire un code source en bas de la page pour le poursuivre sur la page suivante. De même, avoir besoin d'écrire un programme sur plusieurs pages doit alerter le candidat : les réponses attendues dépassent très rarement la demi-page.

3 Commentaires détaillés

Pour chaque question, un tableau récapitule les taux de réussite avec les conventions suivantes :
— 0 signifie qu'aucun point n'a été donné pour la question (question non traitée ou totalement fausse) ;
— < 0.5 signifie que moins de la moitié des points ont été donnés ;
— ≥ 0.5 signifie que plus de la moitié des points ont été donnés ;
— 1 signifie que la totalité des points ont été donnés.

Question 1

0		< 0.5		≥ 0.5		1		Total	
64	11%	13	2%	51	9%	409	76%	537	100%

En question 1, on demandait une fonction de recherche d'un point dans un ensemble de points représenté par une liste. Il s'agissait donc d'effectuer un simple parcours de la liste, de renvoyer `True` dès que le point cherché et le point courant coïncidaient ou bien alors de retourner `False` en cas de parcours infructueux.
Remarques :
— Compte-tenu de la simplicité de cette question, les correcteurs ont sanctionné des algorithmes corrects mais compliqués. Par exemple, certains candidats calculent un booléen qu'ils renvoient après un parcours complet de la liste. D'autres candidats procèdent en deux étapes, filtrant d'abord la liste en ne considérant que les ordonnées puis filtrant dans un second vis-à-vis des abscisses.
— La construction `p in q` n'avait pas été listée dans la section introductive : elle était donc interdite.
— En Python, `'True'` et `True` sont deux valeurs différentes et `None` n'est pas la même chose que `False`.

Question 2

0		< 0.5		≥ 0.5		1		Total	
4	0%	6	1%	76	14%	451	83%	537	100%

Dans cette seconde question, on demandait d'implémenter le calcul de l'intersection entre une liste de points p et une liste de points q en suivant l'algorithme qui consiste à itérer sur p et à accumuler dans le résultat les seuls points de p aussi présents dans q.
Cet algorithme nécessitait l'introduction d'une variable d'accumulation et l'appel à la fonction définie dans la question précédente.
Remarques :
— Ne pas réutiliser la réponse à la question précédente entrainait une légère pénalité.

Question 3

0		< 0.5		≥ 0.5		1		Total	
34	6%	20	3%	76	14%	407	75%	537	100%

La question 3 portait sur la complexité du programme écrit en question 2. La complexité $O(len(p) \times len(q))$ était attendue.

Remarques :

- On exprime les complexités avec la notation de Landau $O(\cdots)$.
- Une complexité de $O(2 \cdot len(p) \cdot len(q))$ n'a pas de sens.
- Comme rappelé dans le sujet sujet, une complexité se justifie en raisonnant sur la structure du code.
- Les paramètres utilisés dans l'expression d'une complexité doivent être définis explicitement. Par exemple, $O(n \cdot m)$ n'a pas de sens si n et m sont indéfinis.

Question 4

0		< 0.5		≥ 0.5		1		Total	
39	7%	6	1%	38	7%	454	84%	537	100%

Après avoir décrit un modèle de données pour représenter la relation d'appartenance d'un point à un ensemble de points, on demandait une requête SQL pour calculer la liste des ensembles dans lequel un point de coordonnées (x, y) données apparaît.

Remarques :

- Il y a un grand nombre d'erreurs de syntaxe dans les réponses des candidats.
- Attention à minimiser la taille de la requête. Par exemple, une requête de la forme

  ```
  SELECT ... FROM (SELECT * FROM ...)
  ```

  est généralement simplifiable.

Question 5

0		< 0.5		≥ 0.5		1		Total	
362	67%	6	1%	28	5%	141	26%	537	100%

En question 5, on attendait une requête SQL pour calculer l'ensemble des points de l'intersection de deux ensembles d'identifiants i et j.

Remarques :

- Les candidats ont souvent écrit des requêtes produisant un résultat vide car imposant la contrainte `idensemble = i AND idensemble = j`.

Question 6

0		< 0.5		≥ 0.5		1		Total	
212	39%	24	4%	125	23%	176	32%	537	100%

On demandait les identifiants des points appartenant à au moins l'un des ensembles où le point de coordonnées (a, b) apparaissait. Il suffisait de réutiliser le résultat R de la requête de la question 1 en cherchant les points parmi les lignes de `MEMBRES` dont l'ensemble figurait dans R.

Remarques :

- Renvoyer les coordonnées des points – et non leurs identifiants comme demandé – a été sanctionné.
- Les candidats ont parfois tendance à rajouter des jointures inutiles. Même en cas de requête correcte, ces jointures inutiles ont entrainé une légère pénalité.

Question 7

0		< 0.5		≥ 0.5		1		Total	
77	14%	3	0%	27	5%	430	80%	537	100%

Après avoir défini le codage de Lebesgue d'un point, on demande au candidat de donner la représentation en Python de celui de $(1, 6)$. Il s'agissait donc d'une application immédiate pour vérifier la compréhension des définitions.

Remarques :

- Il y a eu des étourderies en particulier concernant l'ordre de l'entrelacement des bits.
- Une réponse brute sans justification a été pénalisée : il faut donner les étapes du calcul.
- On demandait une liste Python en réponse et pourtant certains candidats ont répondu à l'aide d'une autre notation.

Question 8

0		< 0.5		≥ 0.5		1		Total	
63	11%	70	13%	191	35%	213	39%	537	100%

Dans cette question, les candidats devaient implémenter la fonction `code(n,p)` qui renvoie le codage du point `p` dans D_n. Il fallait donc itérer sur les représentations binaires des deux coordonnées de `p` en utilisant la fonction `bits` fournie et construire la liste résultat en combinant les bits rencontrés lors de ce parcours.

Remarques :

- Beaucoup de candidats ont réimplémenté la fonction `bits`. C'était hors-sujet.
- Certaines réponses contenaient des conversions inutiles (traduction en base 10 puis retour à la base 2) ou effectuent le calcul en plusieurs passes. Encore une fois, la simplicité et le minimalisme ont été valorisés.
- Les erreurs dans les indices des accès aux tableaux ont été nombreuses dans cette question. Attention à bien vérifier les limites des domaines d'itération des boucles.

Question 9

0		< 0.5		≥ 0.5		1		Total	
10	1%	0	0%	5	0%	522	97%	537	100%

Le sujet introduisait une relation d'ordre lexicographique entre les codages de Lebesgue. La question 9 testait la bonne compréhension de cette définition en demandant au candidat de trier un ensemble de codages par ordre lexicographique croissant.

Remarques :

- La définition de l'ordre lexicographique a parfois été mal comprise.
- Certaines copies n'ont pas respectées la consigne, classant par ordre décroissant plutôt que croissant.

Question 10

0		< 0.5		≥ 0.5		1		Total	
76	14%	54	10%	78	14%	329	61%	537	100%

La question 10 portait sur l'implémentation de l'ordre lexicographique décrit en question 9.

Remarques :

- Encore une fois, les correcteurs ont été étonnés par le nombre de réponses correctes mais inutilement compliquées : par exemple, les programmes effectuant deux passes plutôt qu'une ont été pénalisés. Dans une moindre mesure, un parcours de la totalité des deux listes a été moins bien noté qu'un parcours s'arrêtant dès que la première paire de chiffres distincts est atteinte.
- On trouve aussi des erreurs plus classiques : indices invalides, parcours dans le mauvais sens, algorithme récursif sans cas de base valide.

Question 11

0		< 0.5		≥ 0.5		1		Total	
29	5%	6	1%	13	2%	489	91%	537	100%

Dans cette question, les candidats devaient appliquer les définitions précédentes pour représenter un ensemble de points de $D_n \times D_n$ comme une liste de codages de Lebesgue triée lexicographiquement.
Remarques :

— Les codages ont parfois été classés par ordre décroissant.

Question 12

0		< 0.5		≥ 0.5		1		Total	
84	15%	5	0%	8	1%	440	81%	537	100%

Dans une liste triée de codages de Lebesgue, quatre codages successifs peuvent représenter un quadrant. En se donnant un codage pour les quadrants (ici *via* l'usage du chiffre 4), on peut compacter la représentation d'un espace de points. Il était demandé aux candidats de donner la version compactée de la représentation trouvée dans la question précédente.
Remarques :

— Même si la réponse à la question précédente était fausse, une compaction valide pour cette réponse donnait la totalité des points (à condition bien sûr qu'au moins une compaction soit possible).

Question 13

0		< 0.5		≥ 0.5		1		Total	
78	14%	71	13%	288	53%	100	18%	537	100%

La question portait sur une fonction auxiliaire `ksuffix` qui servait à calculer le quadrant englobant immédiatement un quadrant donné. Il s'agissait d'abord de tester si le codage fournit en entrée était un quadrant de D_k et le cas échéant de renvoyer une nouvelle liste représentant le quadrant englobant de D_{k+1}. Sinon, la fonction devait renvoyer le codage inchangé.
Remarques :

— On a sanctionné les réponses qui modifiaient la liste prise en entrée en oubliant de créér une nouvelle liste, comme l'indiquait la consigne.

— Certains programmes effectuaient des écritures inutiles de 4 à des positions de la liste où un test venait de garantir la présence de 4.

Question 14

0		< 0.5		≥ 0.5		1		Total	
416	77%	68	12%	27	5%	26	4%	537	100%

L'algorithme de compaction d'une liste triée de codages de Lebesgue devait être implémentée en question 14. Cette question était probablement la plus difficile du sujet et un grand nombre de candidats n'a pas trouvé de réponse correcte. Il fallait effectuer plusieurs passes successives sur la liste en compactant les quadrants de D_k avant de compacter les quadrants de D_{k+1}.
Remarques :

— Rares sont les copies correctes et n'ayant pas utilisé la fonction auxiliaire `ksuffix`. On rappelle aux candidats que les réponses aux questions difficiles s'appuient très souvent sur les réponses à des questions préliminaires.

— Même pour les questions difficiles, il est très rare que la réponse attendue ne dépasse la vingtaine de lignes de code : une réponse s'étalant sur plusieurs pages est donc très probablement trop compliquée, et presque sûrement incorrecte.

Question 15

0		< 0.5		≥ 0.5		1		Total	
416	77%	23	4%	17	3%	81	15%	537	100%

Dans cette question, on étendait la relation de comparaison sur les codages de Lebesgue pour prendre en compte la possible inclusion entre les ensembles qu'ils représentent. On pouvait par exemple partir de la réponse à la question 10 en y insérant les cas pour les chiffres valant 4.

Remarques :

— On retrouve les mêmes erreurs qu'en question 10.

— Quelques candidats ont oublié la moitié des cas.

Question 16

0		< 0.5		≥ 0.5		1		Total	
504	93%	17	3%	3	0%	13	2%	537	100%

Pour finir, les candidats devaient programmer la fonction de calcul de l'intersection entre deux ensembles de points représentés par des listes triées et compactées de codages de Lebesgue. L'algorithme suivait le principe de la fusion de deux listes triées : on consomme les deux listes en même temps et on prend une décision sur l'élément à insérer dans la liste résultat en fonction de la comparaison entre les éléments de tête des deux listes. En particulier, quand on a une relation d'inclusion entre les ensembles représentées par ces éléments de tête, on doit insérer le plus petit de ces ensembles dans le résultat.

Remarques :

— Les candidats qui ont traité cette question sans obtenir tous les points ont souvent proposé une structure correcte de fusion de liste mais n'ont pas utilisé les bons critères d'insertion dans la liste résultat.