Banque PC - ENS/X/ESPCI- Session 2017

Rapport sur l'épreuve écrite d'informatique (2h)

Coefficients (en pourcentage du total d'admission) :
Lyon : 5,3%
Paris-Saclay : 5,1%
Ecole Polytechnique et ESPCI : 2,9%

Membres du jury : Florian HORN, Yann REGIS-GIANAS

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.