ÉCOLE POLYTECHNIQUE CONCOURS D'ADMISSION 2018
ENS : CACHAN, LYON, RENNES

Composition d'Informatique 2h, Filière MP (XLCR)

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 555. La note moyenne est de 11,1 avec un écart-type de 3,2. La note minimale est de 1,9/20 et la note maximale 19,1/20. Aucune copie n'a obtenu une note éliminatoire. Au total, 24 candidats (4,3%) ont traité le sujet dans son intégralité, c'est-à-dire ont obtenu une note strictement positive à chacune des 26 questions. La répartition des notes est résumée dans le tableau suivant :

	[0;4[	[4;8[	[8;12[	[12;16[	[16;20]
Effectif	3 (0,6%)	111 (20%)	216 (38,9%)	191 (34,4%)	34 (6,1%)

La répartition des notes des candidats français de l'école polytechnique est résumée dans le tableau suivant :

0<=N<4	1	0,51%
4<=N<8	63	31,82%
8<=N<12	97	48,99%
12<=N<16	31	15,66%
16<=N<=20	6	3,03%
Total :	198	100%
Nombre de copies :	198	
Note moyenne :	9,61	
Ecart-type :	2,82	

2 Commentaires

Le sujet portait cette année sur l'implémentation de l'algèbre relationnelle. La première partie portait sur une implémentation naïve en Python des opérateurs de cette algèbre. La deuxième partie consistait en une mise en pratique des opérateurs ainsi définis, pour exprimer des requêtes SQL. La troisième partie proposait d'optimiser l'implémentation naïve de deux manières, en conservant les tables triées d'une part, et en utilisant un dictionnaire pour effectuer une indexation d'autre part.

Comme chaque année, nous rappelons que la présentation des solutions doit être soignée. En particulier, l'indentation est un élément de syntaxe du langage Python, le niveau d'indentation de chaque ligne de code doit être lisible sans ambiguïté. Pour lever tout doute, il est possible de matérialiser chaque niveau d'indentation par un trait vertical continu.

Il est rappelé l'importance de lire le sujet dans son intégralité avant de traiter les questions, et de respecter scrupuleusement les consignes, on peut citer entre autres l'utilisation non autorisée :

- des tranches de liste (syntaxe de type `l[i:j]`)
- de la génération de liste utilisant une syntaxe de type `[f(x) for x in … if …]`,
- de l'opérateur binaire `in`,
- de la fonction pop avec un indice en paramètre, seul l'appel sans paramètre était autorisé,
- des fonctions enumerate ou zip.

La mention de l'égalité de liste comme étant interdite n'était qu'un exemple. Tout ce qui n'était pas explicitement autorisé ne pouvait être utilisé. Cela a posé d'autant plus problème que leur complexité n'était ainsi pas supposée connue.

De nombreuses confusions ont été également constatées entre l'ajout d'un élément à la fin d'une liste avec append() et la concaténation avec +. « `l+=e` » n'est pas équivalent à « `l.append(e)` ».

Lorsque l'énoncé préconise d'implémenter une fonction auxiliaire, il est recommandé de suivre cette préconisation. On peut citer l'exemple de la question I.7 : la fonction auxiliaire, que l'on aurait pu appeler JointureEnregistrement, est utile pour cette question, mais aussi pour les questions III.3 et III.8. Lors de l'introduction de fonctions auxiliaires non suggérées par le sujet, il vaut mieux leur donner un nom explicite plutôt que devoir recourir à de longues explications. De manière générale, il vaut mieux écrire des fonctions simples avec des noms de variables adaptés plutôt qu'avoir besoin d'expliquer un fonctionnement complexe. Il n'est utile de complexifier l'implémentation que si la complexité algorithmique est réellement différente ou que par exemple elle est facilement divisée en moyenne par deux.

Une fonction ne devrait pas modifier « en place » les paramètres qui lui sont passés lorsque cela ne fait pas partie de sa spécification.

Il est préférable d'éviter des tests inutilement compliqués tels que « `if not a == b:` »

Il est en général inutile de traiter à part le cas d'une liste vide (ou, le cas échéant, à un élément), les boucles s'accordent naturellement avec ce cas.

On rappelle qu'il est possible d'itérer sur les éléments 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, plus lisible, et évite des erreurs de décalage d'indice.

Lorsque l'on itère sur un indice de liste, il faut vérifier que cet indice n'a pas dépassé la longueur de la liste avant de l'utiliser pour accéder à l'élément correspondant : « `while t[i] == x and i < len(t):` » provoque un dépassement de liste.

3 Commentaires détaillés

Pour chaque question, on présente la répartition des résultats suivant les quatre catégories suivantes :

- 0 si la question est non traitée ou la réponse est complètement fausse,

-]0;0,5[si la question est partiellement traitée ou la réponse comporte de nombreuses erreurs,
- [0,5;1[si la question est traitée correctement avec quelques erreurs,
- 1 si la question est traitée correctement.

Question I.1

0	]0;0,5[	[0,5;1[	1
21 (3,8%)	0 (0%)	88 (15,9%)	446 (80,4%)

Itérer directement sur les éléments de la liste permet d'obtenir une version bien plus simple qu'avec l'utilisation d'indices.

Question I.2

0	]0;0,5[	[0,5;1[	1
8 (1,4%)	0 (0%)	3 (0,5%)	544 (98%)

Il est nécessaire de justifier un minimum la complexité proposée.

Question I.3

0	]0;0,5[	[0,5;1[	1
16 (2,9%)	0 (0%)	19 (3,4%)	520 (93,7%)

De même, cette fonction peut s'écrire très simplement en itérant sur les éléments. Attention à la confusion entre ajout d'un élément et concaténation de listes.

Question I.4

0	]0;0,5[	[0,5;1[	1
33 (5,9%)	0 (0%)	35 (6,3%)	487 (87,7%)

L'implémentation itérant sur les éléments était ici particulièrement simple. La liste d'indices étant déjà dans l'ordre voulu, il était inutile de chercher une implémentation plus complexe.

Question I.5

0	]0;0,5[	[0,5;1[	1
21 (3,8%)	0 (0%)	21 (3,8%)	513 (92,4%)

De nouveau il est nécessaire de ne pas confondre ajout d'un élément à la fin de la liste et concaténation de listes.

Question I.6

0	]0;0,5[	[0,5;1[	1
13 (2,3%)	0 (0%)	15 (2,7%)	527 (95%)

Il était ici utile de profiter de la concaténation de liste, autorisée explicitement par le sujet.

Question I.7

0	]0;0,5[	[0,5;1[	1
16 (2,9%)	24 (4,3%)	232 (41,8%)	283 (51%)

Il n'était pas très intéressant de fabriquer une liste d'indices et d'appeler ProjectionEnregistrement. La complexité est la même, mais cela nécessite de construire une liste d'indices assez triviales plutôt qu'utiliser une simple boucle for.

Il n'était pas avisé de placer le test d'égalité dans la fonction JointureEnregistrement, car Jointure doit alors de nouveau tester le résultat de JointureEnregistrement.

Il a parfois été proposé une implémentation utilisant ProduitCartesien et SelectionEgalité. D'une part, il faut aussi utiliser une Projection pour joindre réellement les deux indices. D'autre part, même si la complexité est la même, implémenter directement la construction de la liste est bien moins coûteux en mémoire lorsque le résultat est bien plus petit que le produit cartésien complet.

Il était nécessaire de s'assurer de fabriquer une nouvelle liste en concaténant par exemple une liste vide à la liste passée en paramètre.

Question I.8

0	]0;0,5[	[0,5;1[	1
78 (14,1%)	0 (0%)	53 (9,5%)	424 (76,4%)

Il était bien sûr nécessaire de penser à prendre en compte le coût des fonctions auxiliaires appelées.

Question I.9

0	]0;0,5[	[0,5;1[	1
110 (19,8%)	37 (6,7%)	221 (39,8%)	187 (33,7%)

Malgré le nom indiqué, la meilleure approche n'était pas d'enlever des éléments de la table à l'aide de del ou de pop(i), qui n'étaient de toutes façons pas autorisées, et qui mènent à une approche bien trop coûteuse (del et pop(i) de Python sont en O(len(L))), mais de reconstruire une table exempte de doublons.

Même si ce n'était pas suggéré, il était évident qu'utiliser une ou plusieurs fonctions intermédiaires rendait l'écriture bien plus facile (et ainsi, correcte et efficace...)

Il vaut mieux tester la présence de l'élément dans la table résultat, qui est au pire de la même taille que la table d'origine, au mieux bien plus petite. Sinon, il faut prendre garder à ne pas comparer la ligne à elle-même lors de la détection d'un doublon.

Il vaut mieux utiliser des returns dans des fonctions auxiliaires pour diminuer le coût lorsque des éléments identiques sont trouvés. Un return bien placé est par ailleurs souvent bien plus lisible qu'une boucle while.

Il n'y avait pas lieu d'essayer d'introduire une version récursive, moins simple à écrire, et dont la complexité est bien moins simple à évaluer. Il est par ailleurs problématique que l'implémentation proposée ne soit pas récursive terminale.

Question I.10

0	]0;0,5[	[0,5;1[	1
100 (18%)	0 (0%)	45 (8,1%)	410 (73,9%)

Il est nécessaire de s’assurer de la complexité des opérateurs utilisés, indiquée dans le sujet. Très souvent, les opérateurs utilisés à tort car non autorisés ont été considérés comme ayant un coût constant, alors qu’ils ont justement un coût linéaire.

Question II.1

0	]0;0,5[	[0,5;1[	1
18 (3,2%)	0 (0%)	90 (16,2%)	447 (80,5%)

Il est nécessaire de respecter strictement la syntaxe Python, et notamment utiliser des apostrophes pour délimiter les chaînes de caractères.

Question II.2

0	]0;0,5[	[0,5;1[	1
27 (4,9%)	0 (0%)	1 (0,2%)	527 (95%)

Cette question n’a pas posé de problème particulier.

Question II.3

0	]0;0,5[	[0,5;1[	1
40 (7,2%)	0 (0%)	35 (6,3%)	480 (86,5%)

Attention le résultat d'une jointure n'est pas exactement le même que le résultat d'un produit cartésien avec une sélection, c'était la deuxième solution qui était attendue. Les jointures ont tout de même été acceptées.

Question II.4

0	]0;0,5[	[0,5;1[	1
47 (8,5%)	0 (0%)	52 (9,4%)	456 (82,2%)

Cette fois-ci c'était une jointure qui était attendue. Il était nécessaire de faire bien attention aux indices pour la projection.

Question II.5

0	]0;0,5[	[0,5;1[	1
47 (8,5%)	144 (25,9%)	314 (56,6%)	50 (9%)

Pour limiter l'explosion combinatoire, il est préférable de sélectionner d'abord les tickets de prix 50, et d'utiliser des jointures plutôt que des produits cartésiens. Il ne faut pas oublier de finir par une projection.

Même si l'on ne passe qu'un seul indice à la projection, il faut l'inclure dans une liste.

Il vaut mieux écrire le programme sur plusieurs lignes pour une lisibilité bien meilleure.

Question II.6

0	]0;0,5[	[0,5;1[	1
201 (36,2%)	46 (8,3%)	245 (44,1%)	63 (11,4%)

Il était bien sûr plus qu'utile de remarquer que l'on pouvait réutiliser le résultat de la question précédente.

De nouveau, il était intéressant d'utiliser une sélection et une jointure. La projection finale était par contre inutile.

Question III.1

0	]0;0,5[	[0,5;1[	1
110 (19,8%)	0 (0%)	123 (22,2%)	322 (58%)

Les solutions quadratiques étaient bien sûr exclues. Il faut bien sûr penser à effectuer un tour de boucle de moins qu'il y a d'éléments dans la liste.

la solution qui consiste à trier la liste, puis vérifier que la liste triée est identique à la liste originale n'est pas acceptable. Au-delà de la complexité (au mieux en O(n log n) pour le tri, alors qu'il est possible de vérifier que la liste est triée en O(n)), il peut exister plusieurs versions valides de la liste triée en cas d'égalité selon un des critères, on ne peut donc pas conclure avec cette méthode.

Il était admis que l'opérateur de comparaison python pouvait être utilisé pour comparer deux chaines de caractères. Dès lors, il était inutile de chercher à re-coder une fonction de comparaison pour l'ordre lexicographique.

Question III.2

0	]0;0,5[	[0,5;1[	1
88 (15,9%)	15 (2,7%)	93 (16,8%)	359 (64,7%)

Bien sûr l'utilisation d'une dichotomie apporte la meilleure complexité, mais elle n'est pas immédiate à écrire, car ce n'est pas un seul élément que l'on recherche. Il vaut mieux commencer par proposer une solution simple mais fonctionnelle, puis s'atteler à essayer de proposer une dichotomie.

Il était possible, comme version plutôt simple, d'utiliser une dichotomie pour trouver un élément ayant la valeur recherchée, puis de là utiliser des recherches linéaires pour trouver tous les autres.

Dans une approche dichotomique, il faut faire attention aux indices, utiliser m+1 comme indice peut éventuellement déborder du tableau.

Par ailleurs, il est indispensable d'expliciter les invariants utilisés.

Enfin, il est utile de placer l'implémentation de la dichotomie dans une fonction auxiliaire pour pouvoir la réutiliser dans la question suivante.

Question III.3

0	]0;0,5[	[0,5;1[	1
263 (47,4%)	24 (4,3%)	117 (21,1%)	151 (27,2%)

Il était possible d’utiliser de nouveau une dichotomie. Il était également possible de parcourir les deux tables en parallèle, de manière similaire à un tri fusion, plutôt que repartir à chaque fois du début de la deuxième table parcourue.

Question III.4

0	]0;0,5[	[0,5;1[	1
328 (59,1%)	11 (2%)	97 (17,5%)	119 (21,4%)

Il est essentiel de ne pas négliger la complexité des fonctions auxiliaires. Il faut réellement montrer dans quels cas on constate une amélioration par rapport à l’implémentation naïve.

Question III.5

0	]0;0,5[	[0,5;1[	1
230 (41,4%)	7 (1,3%)	54 (9,7%)	264 (47,6%)

Il faut penser à traiter à part le cas où le dictionnaire ne comporte pas encore d'entrée pour la valeur à ajouter.

Il n'est pas intéressant de construire explicitement la liste des éléments ayant la même valeur: puisque l'accès au dictionnaire est en temps constant, il est très peu cher d'ajouter un élément à une liste précédemment ajoutée au dictionnaire.

Cette fois-ci, il était plus judicieux d'itérer sur les indices que sur les éléments. Dans le second cas, il est en effet nécessaire de maintenir « à la main » l'indice de l'élément courant, avec risque d'erreur.

Question III.6

0	]0;0,5[	[0,5;1[	1
256 (46,1%)	12 (2,2%)	153 (27,6%)	134 (24,1%)

De nouveau, le dictionnaire ne contient pas nécessairement la valeur recherchée, il est nécessaire de tester ce cas-là.

Question III.7

0	]0;0,5[	[0,5;1[	1
296 (53,3%)	7 (1,3%)	51 (9,2%)	201 (36,2%)

De nouveau il était essentiel de discuter sur les exemples demandés pour montrer la réelle compréhension de la réduction de complexité.

Question III.8

0	]0;0,5[	[0,5;1[	1
355 (64%)	7 (1,3%)	97 (17,5%)	96 (17,3%)

Il était possible de réutiliser la fonction auxiliaire proposée à la question I.7, ce qui ramène à itérer simplement sur la table1 et le dico2.

Question III.9

0	]0;0,5[	[0,5;1[	1
396 (71,4%)	0 (0%)	43 (7,7%)	116 (20,9%)

Il était intéressant de noter que la longueur des listes renvoyées par le dictionnaire est en général bien plus petite que la longueur de la table, apportant donc un gain appréciable.

Question III.10

0	]0;0,5[	[0,5;1[	1
443 (79,8%)	0 (0%)	41 (7,4%)	71 (12,8%)

La complexité étant le produit de deux grandeurs, il fallait se demander laquelle était la plus intéressante à minimiser. Certains candidats ont pensé à considérer la complexité plus globalement, plus finement qu'avec la seule grandeur k_2. Le nombre total de tours de la boucle la plus interne est en effet égal à la longueur du résultat dans les deux cas, le choix d'indexer table1 ou table2 ne change donc pas son coût. Il change par contre le coût de la boucle externe.