École Polytechnique — Concours d'admission 2020
ENS Ulm, Lyon, Paris-Saclay, Rennes

Composition d'Informatique – A – (XULCR)

filière MP spécialité Info

1 L'épreuve

Il s'agissait dans cette épreuve d'étudier la notion de labyrinthe, défini comme un sous-graphe d'un graphe non-orienté et partageant le même ensemble de sommets. Une classe de labyrinthes particulière, les labyrinthes parfaits (connexes et acycliques), était particulièrement mise en avant.

La première partie présentait la notion de labyrinthe et s'intéressait aux méthodes de génération aléatoires de labyrinthes. Trois algorithmes de génération étaient étudiés : un parcours en profondeur avec mélange aléatoire des listes d'adjacence et deux algorithmes basés sur des classes d'équivalence (union-find), dont l'algorithme d'Eller.

La deuxième partie abordait la résolution de labyrinthes, c'est-à-dire la recherche d'un chemin de longueur minimale entre un sommet source et un sommet destination. Une preuve de programme basé sur un parcours en largeur était demandée. Une variante était proposée, avec la présence de monstres sur les sommets : il s'agissait alors de rechercher un chemin le plus court parmi ceux passant par un minimum de monstres.

Les notes des 1127 copies se répartissent selon le tableau suivant, avec une moyenne de 9,28 et un écart type de 4,17. Pour obtenir la note maximale, il n'était pas nécessaire de traiter l'intégralité du sujet.

$0 \leq N < 4$	126	11,2 %
$4 \leq N < 8$	291	25,8 %
$8 \leq N < 12$	407	36,1 %
$12 \leq N < 16$	233	20,7 %
$16 \leq N \leq 20$	70	6,2 %
Nombre de copies	1127	100 %
Note moyenne	9,28	
Écart-type	4,17	

candidats français :

$0 \leq N < 4$	69	8,3 %
$4 \leq N < 8$	208	25,2 %
$8 \leq N < 12$	298	36,0 %
$12 \leq N < 16$	195	23,6 %
$16 \leq N \leq 20$	57	6,9 %
Nombre de copies	827	100 %
Note moyenne	9,70	
Écart-type	4,07	

2 Remarques générales

Clarté des copies et des démonstrations. Trop de copies s'avèrent difficiles à lire pour les correcteurs. Les candidats qui produisent des réponses raturées, mal indentées ou encore non structurées en paragraphe perdent nécessairement des points malgré tous les efforts de décodage des correcteurs. Les centres d'examen fournissent des feuilles de brouillon aux candidats.

On rappelle que pour être valide une preuve de programme doit suivre avec précision la structure du programme. Trop souvent, les candidats utilisent une récurrence sur la taille d'une structure de données plutôt qu'un schéma de preuve par invariants, comme le suggérait pourtant le sujet. Par ailleurs, les preuves par invariants ne sont pas du tout maîtrisées par un grand nombre de candidats. Pour les questions sur la correction d'un algorithme, on attend autre chose qu'une simple paraphrase de cet algorithme.

Erreurs dans les programmes. On rappelle qu'on parcourt les éléments d'un tableau de taille `n` avec une boucle `for` en écrivant `for i = 0 to n - 1 do` et non pas `for i = 0 to n do`. Cette erreur a été vue dans de trop nombreuses copies.

De trop nombreuses erreurs de sémantique rendent incorrects les programmes proposés par les candidats. Par exemple : `let x = 4 in match y with x => ...` n'est pas équivalent à `if y = 4 then ...`. Les erreurs de typage mettent aussi à mal la correction des programmes : en OCaml, les booléens ne sont pas des entiers, les tableaux ne sont pas des listes, etc.

On souhaite aussi alerter les candidats sur l'influence de Python dans l'écriture de programmes OCaml. Les candidats ont tendance à oublier le caractère immuable des listes OCaml : écrire `x :: l` ne modifie pas la liste `l` mais représente la liste formée par la valeur de `x` suivie de la valeur de `l`. Enfin, la syntaxe d'OCaml est très différente de celle de Python : l'indentation n'est pas significative (et donc l'oubli de `begin-end` ou de parenthèses peut être fatal), les itérations de la forme `for i in range ...` n'existent pas, les affectations multiples non plus, etc.

Concision des programmes. Le sujet indiquait que toute fonction de la bibliothèque standard d'OCaml pouvait être utilisée et donnait aussi des fonctions prédéfinies (`graphe_vide`, `ajoute_arete`, etc.) pour faire gagner du temps aux candidats : les copies qui n'ont pas utilisées ces fonctions n'ont pas été sanctionnées mais se sont sanctionnées elles-mêmes en perdant du temps à les réimplémenter. D'une manière générale, les programmes attendus sont souvent courts : un candidat qui écrit un long programme avec un haut niveau d'imbrications et de cas doit s'interroger sur l'existence d'une solution plus simple.

On regrettera aussi l'usage de constructions lourdes là où une autre construction syntaxique équivalente existe : `match x with x when x <> y =>` à la place de `if x <> y then` ou bien `match e with (y, z) =>` à la place de `let (y, z) = e in` ou encore `if cond then () else` à la place de `if not cond then`.

De même, il est regrettable que certains candidats recourent systématiquement à des fonctions récursives là où une simple boucle `for` est plus concise (par exemple, le parcours des voisins dans la question 3) ou, inversement, recourent systématiquement à une boucle là où une fonction récursive est plus concise (par exemple à la question 5).

3 Commentaire détaillé

Pour chaque question, sont indiqués entre crochets le pourcentage de candidat(e)s ayant traité la question et le pourcentage de candidat(e)s ayant obtenu la totalité des points.

Partie I

Question 1 [100% - 74%]. Cette question ne présentait pas de difficultés particulières. On s'étonne quand même de nombreuses erreurs sur les bornes et dans l'échange de deux valeurs dans un tableau.

Question 2 [96% - 35%]. De nombreux candidats présentent des disjonctions de cas incomplètes ou déclarent tous les cas triviaux sauf un. Ces erreurs ont été lourdement sanctionnées.

Question 3 [90% - 15%]. Des candidats utilisent une liste plutôt qu'un tableau pour se rappeler des sommets déjà explorés. Même si aucune complexité particulière n'était exigée ici, c'est tout de même une solution très inefficace. Le sujet était fait pour qu'un simple tableau fournisse une solution simple et efficace au marquage des sommets.

Il fallait être attentif ici aux types des fonctions : la fonction `melange_knuth` attendait un tableau, pas une liste, et ne renvoyait pas le tableau mélangé mais effectuait son traitement en place.

Trop de codes sont inutilement compliqués, avec les risques d'erreur que cela implique. On a vu par exemple des fonctions récursives avec trois listes de sommets en arguments. Certaines copies explicitent une structure de pile pour écrire le un parcours en profondeur. Cela est tout à fait correct, bien entendu, mais inutile ici : un éventuel débordement de la pile d'appel n'est pas sanctionné dans une copie de concours.

Question 4 [65% - 2%]. La notion de labyrinthe parfait est incomprise par de nombreux candidats, qui ne vérifient que l'acyclicité, oubliant la connexité. Par ailleurs, l'argument essentiel consistant à faire remarquer qu'un sommet non encore visité n'est pas encore relié au labyrinthe en cours de construction est absent de la plupart des copies.

Question 5 [99% - 74%]. C'est ici un exemple de code qui s'écrivait très facilement avec une fonction récursive, en une ligne. Certains candidats ne suivent pas les consignes ici et supposent un tableau en argument, plutôt que la structure de classes disjointes.

Question 6 [97% - 38%]. La seule difficulté était de bien penser à mettre à jour le rang de la nouvelle classe lorsque les rangs des deux classes fusionnées sont égaux. Certains candidats ont oublié de s'assurer que l'union était faite sur deux classes disjointes.

Question 7 [87% - 32%]. Certains candidats ne comprennent pas ce que veut dire « préserve l'invariant » et donnent un exemple pour lequel on a moins que 2^r éléments dans une classe de rang r (évidemment pas construite par itérations de la fonction `cd_union`). De nombreux candidats ne justifient pas correctement la taille du plus long chemin et se contente de donner une borne inférieure sur cette valeur.

Cette question 7 permettait aux candidats d'éventuellement corriger leur question 6. Un nombre surprenant de candidats prouvent que les invariants sont satisfaits pour une version correcte de la mise à jour du rang mais n'en profitent pas pour corriger leur code de la question 6.

Question 8 [76% - 52%]. On attendait ici un argument rapide par récurrence pour affirmer que les classes de la relation d'équivalence vérifient bien les invariants de la question précédente.

Question 9 [91% - 31%]. Cette question demandait un peu de soin pour programmer un algorithme décrit dans le sujet. Comme à la question 3, il fallait être attentif au type de la fonction `melange_knuth`.

Question 10 [63% - 12%]. On a vu ici trop de preuves vagues et peu convaincantes, en particulier par l'absurde. Parmi les preuves par invariants, plusieurs n'utilisent même pas l'invariant pour justifier qu'il y a unicité au sein de chaque composante connexe avant la fusion. Trop de candidats oublient des cas lors de la fusion de deux classes X et Y : il existe des chemins entre X et Y mais aussi au sein des (anciennes) classes X et Y.

Question 11 [46% - 10%]. La plupart des candidats qui traitent cette question ne montrent que l'acyclicité du sous-graphe obtenu. La principale difficulté de cette question résidait pourtant dans la preuve de la connexité du résultat, qui s'obtenait comme conséquence des étapes 1b et 2.

Question 12 [26% - 2%]. Cette question a été peu abordée et rarement réussie. La difficulté consistait à montrer avec rigueur que les sous-graphes accessibles à l'algorithme d'Eller peuvent correspondre exactement aux sous-graphes des labyrinthes parfaits.

Partie II

Question 13 [97% - 34%]. Cette question comportant un petit piège : dans la majorité des langages de programmation, et en OCaml en particulier, le modulo d'un entier négatif est négatif. Dès lors, on ne pouvait se contenter d'écrire simplement `(f.debut - 1) mod cap`. Cette erreur n'a cependant pas été lourdement sanctionnée. Beaucoup de copies oublient de mettre à jour le taille `taille`.

Question 14 [96% - 34%]. Il ne fallait pas oublier de renvoyer la valeur retirée de la file. Dans certaines copies, la valeur renvoyée n'apparaît pas à la fin du flot de contrôle du programme, ce qui est bien évidemment une erreur.

Question 15 [77% - 3%]. Il est surprenant que si peu de copies traitent correctement cette question, alors que le parcours en largeur est au programme. La question a pu dérouter certains candidats car la mise en œuvre du parcours en largeur décrite dans le sujet n'était sans doute pas celle qu'ils avaient vue en cours. Il est très surprenant que certains de ces candidats aient donné, sans sourciller, la preuve correspondant à une autre mise en œuvre du parcours en largeur, qui était évidemment fausse. La preuve doit explicitement faire référence à l'algorithme que l'on considère.

Question 16 [83% - 26%]. On ne demandait pas aux candidats de recopier l'intégralité du code donné dans l'énoncé, mais seulement de compléter les parties manquantes. On demandait encore moins de modifier le code de l'énoncé !

Question 17 [69% - 33%]. On préférera toujours un contre-exemple minimal ou du moins le plus petit possible. Le contre-exemple devait être accompagné d'un minimum de justification. Il fallait également penser à indiquer les sommets source et destination.

Question 18 [64% - 22%]. Cette question d'application demandait une certaine rigueur. Elle avait pour objectif de se familiariser avec l'algorithme.

Question 19 [25% - 0%]. L'énoncé demandait uniquement d'énoncer les invariants permettant de prouver le résultat : il n'était donc pas très utile d'écrire des pages de paraphrase du code. Très peu de candidats savent énoncer proprement des invariants de boucle, même inexacts (malgré les exemples d'invariants donnés précédemment dans l'énoncé).

Question 20 [26% - 5%]. De nombreux candidats proposent une idée pertinente, mais omettent de la justifier. Un énoncé commençant par « Montrer que » attend en général un minimum de justification. On demandait à la fois une définition précise de la réduction et une preuve de sa correction.