systématiquement `for i in range n` ou `len L` ou lors de l'utilisation d'une fonction déjà codée) a été sanctionnée.

- Le sujet demandait explicitement de manipuler des listes imbriquées. Les candidats doivent maîtriser la manipulation des listes, notamment :
  - la construction d'une liste élément par élément. Par exemple, l'initialisation d'une liste `L = []` suivie, dans une boucle `for`, d'une affectation `L[i] = elt` provoque une erreur. De même, l'instruction `L=h*[]` initialise la liste `L` à une liste vide (et pas à une liste de h listes vides). Il en va de même pour des listes formées de listes vides : les syntaxes `[[]] * n` ou `[[]*n]` ne conviennent pas.
  - Le parcours d'une liste dans une boucle `for` peut se faire éléments par éléments (`for elt in L:`) ou indice par indice (`for i in range(L):`) ; la première donnait souvent lieu à des codes plus lisibles. Attention toutefois à ne pas confondre les deux syntaxes.
  - l'ajout d'un élément à la fin d'une liste. Comme indiqué dans les rapports des années précédentes, la syntaxe `L.append(elt)` est à privilégier. D'une part, elle est plus efficace, mais elle est également moins source d'erreurs. L'emploi de la syntaxe `L = L + [elt]` (ou `L += [elt]`) a par exemple provoqué beaucoup d'oublis de crochets, quand `elt` était elle-même une liste.
  - la syntaxe du « slicing » des listes (pour ceux qui ont voulu l'utiliser) n'est pas toujours maîtrisée, notamment en ce qui concerne l'indice final ou l'utilisation des « : » (confondus avec des virgules).
  - le caractère modifiable des listes en Python n'est pas compris par tous. Sauf mention du contraire, les listes rentrées en paramètres des fonctions ne doivent pas être modifiées.

4.1.3 Conseils aux futurs candidats

Nous conseillons aux futurs candidats une lecture attentive du rapport du jury ainsi que du programme officiel d'informatique commune : celui-ci peut aider à vérifier la maîtrise des points exigibles aux concours. De plus, un nombre souvent faible de questions a été traité dans de nombreuses copies, montrant un manque d'entraînement à écrire des codes, mêmes simples. Un investissement un peu plus important des candidats en informatique commune produirait certainement une nette amélioration.

4.2 Informatique option MP

4.2.1 Généralités

Le sujet s'intéresse à la résolution du jeu de hanjie, un jeu de réflexion consistant en la création d'une image sur une grille de pixels noirs ou blancs en utilisant des contraintes indiquées sur chaque ligne et colonne. Ces contraintes précisent les séries de cases noires consécutives dans la ligne ou la colonne et permettent de trouver de façon unique comment colorier la grille.
Le sujet est composé de trois parties indépendantes :

- une première partie s'intéresse à la résolution du hanjie en modélisant le problème en logique propositionnelle (chaque case de la grille est représentée par une variable propositionnelle dont la valeur de vérité spécifie si la case est coloriée en noir ou pas) et en utilisant le système formel de la déduction naturelle pour établir des conséquences logiques des contraintes données. Cette première partie utilise un exemple simple de hanjie sur une grille 2×3.

  Cette partie a été globalement mal traitée par les candidats :

  - beaucoup de candidats ne savent pas écrire correctement une forme normale. En particulier, beaucoup écrivent des formules avec des négations qui s'appliquent sur des formules non atomiques ou mélangent disjonctions et conjonctions.
  - certains candidats ne savent pas non plus écrire une table de vérité correctement. On trouve en particulier des tables de vérité à « double entrée » qui sont difficiles à lire.
  - peu de candidats écrivent des arbres de preuve corrects en déduction naturelle, les questions correspondantes étant souvent très peu traitées. On peut toutefois noter que la déduction naturelle a fait son apparition dans le nouveau programme, ce qui peut expliquer le peu d'aisance des candidats.

- la deuxième partie s'intéresse à la résolution du jeu via une technique algorithmique de retour sur trace. Cette partie de programmation en OCaml était plus « classique » et est a été globalement mieux traitée par les candidats.

  On trouve toutefois encore beaucoup de copies qui mélange les styles de programmation fonctionnel et impératif et dans lesquelles les candidats écrivent des fonctions qui ne font pas ce qui est attendu. Par exemple, le motif de fonction suivant a été souvent vu dans les copies :

```
let ma_fonction(tab : int array) : bool =
  for index = 1 to n do
    if condition_sur tab.(index) then
      false ;
  done ;
  true ;
```

  En dehors du fait que ce code ne compilerait pas (pas de branche `then` donc contradiction avec le type de `false` qui devrait être `unit` et bloc `for` dont le type est systématiquement `unit`), cette fonction renverrait systématiquement `true`.

- la troisième partie permet d'accélérer la résolution du jeu en utilisant une technique de parcours de graphe sur des automates. Elle permet de compléter la stratégie naïve d'extension d'une grille incomplète vue dans la partie 2 en tenant compte des déductions que l'on peut faire. Cette partie était clairement plus difficile et faisait essentiellement appel à des notions de théories des langages, en particulier sur les automates, même si du code OCaml était demandé en fin de partie.

Une analyse détaillée des questions est présentée dans l'annexe S.

4.2.2 Analyse de forme

La plupart des copies proposent du code lisible et indenté, avec des noms de variables et de fonctions compréhensibles. De la même façon, les démonstrations demandées étaient souvent bien présentées lorsqu'elles étaient faites.
On trouve toutefois des copies difficilement lisibles avec du code sans indentation, des renvois en bas de page, des ratures. Ces copies sont très difficilement interprétables par les correcteurs.
Nous rappelons aux candidats :

- qu'il vaut mieux barrer entièrement une réponse et la reprendre plutôt que d'insérer des modifications à l'intérieur de celle-ci ;
- que l'utilisation de fonctions auxiliaires/intermédiaires pertinentes permet de faciliter la lecture et la compréhension de leur code. De la même façon, des commentaires et/ou des noms de variables intelligibles facilitent l'écriture d'un code correct et sa vérification ;
- les expressions comme « il est trivial que », « il est évident que », etc. ne sont pas acceptées lorsque la question attend une justification ou une démonstration (et la justification n'est généralement pas si triviale).

4.3 Informatique 1 filière MPI

4.3.1 Remarques générales

Le sujet s'intéresse à une structure de donnée appelée *liste à accès direct* et étudie différents aspects de sa représentation. Il est composé de 3 parties indépendantes comprenant au total 32 questions.
Le sujet est assez long et nécessite d'aller à l'essentiel sur certaines questions pour ne pas perdre trop de temps.
Le sujet comporte des questions d'analyse, de formalisation et de démonstration, ainsi que des questions de programmation en C. Les candidats ont abordé de façon équilibrée les questions de programmation ainsi que les questions portant sur des démonstrations.
De manière générale, les candidats les plus à l'aise répondent avec précision aux questions posées avec des solutions le plus souvent simples (démonstrations succinctes ou quelques lignes de code). Nous rappelons que les programmes demandés peuvent dans la plupart des cas être écrits en quelques lignes. Il est souvent contre productif d'écrire des codes longs, complexes, avec de multiples tests, qui contiennent presque toujours des erreurs. Il faut privilégier la correction et la simplicité des codes.

4.3.2 Programmation

- Il était inutile d'inclure les bibliothèques standards, comme `<stdio.h>`, `<stdlib.h>`, `<assert.h>` ou `<stdbool.h>`.

R Informatique option MP

Q1 - question bien traitée dans la plupart des cas. On attendait une justification de la même forme que celle présentée dans la description du jeu dans le sujet.
Q2 - certaines formes de tables de vérité étaient surprenantes et difficiles à interpréter. Le plus simple est de créer un tableau où chaque ligne représente une interprétation propositionnelle et les colonnes les valuations des variables et des formules demandées.
Un nombre non négligeable de candidats ne trouvaient pas les interprétations demandées ou écrivaient des formules qui n'étaient pas sous forme normale.
Q3 - mêmes remarques que pour la question précédente.
Q4 - question qui a été globalement non abordée. Les candidats qui l'ont abordée l'ont plutôt bien résolue. L'arbre de preuve demandé était obtenu de façon assez directe.
Q5 - même remarque que la question précédente.
Q6 - question globalement mal traitée. On attendait un argument citant au moins la correction de la déduction naturelle pour justifier la non existence d'un arbre de preuve.
Q7 - question globalement bien traitée.
Q8 - question globalement bien traitée. On note toutefois une confusion entre les opérateurs OCaml (`mod`, `/`) et les opérateurs Python (`%`, `//`) dans un nombre non négligeable de copies.
Q9 - question globalement bien traitée. La principale difficulté consistait à bien faire une copie profonde de l'argument de type `presolution` passé en paramètre. Or `presolution` est un alias pour `couleur array array` ce qui implique qu'il faut

- soit utiliser `presolution_init` pour créer un instance fraîche de `presolution` et la remplir avec les valeurs de `p` sauf pour le `z` passé en paramètre ;
- soit utiliser `Array.copy` mais pas simplement sur `p` sinon la copie n'est pas profonde.

Certains candidats oublient de donner comme valeur « finale » à la fonction l'instance de `presolution` qu'ils ont créée.
Q10 - question globalement mal traitée. Le terme « immuable » est très souvent mal défini et l'avantage demandé est dans la plupart des réponses faux (par exemple, on trouve souvent une amélioration de la complexité temporelle ou spatiale des algorithmes comme avantage).
Q11 - à partir de cette question, il était clairement stipulé dans l'énoncé qu'il fallait manipuler le type `presolution` à travers l'accesseur `get` et le transformateur `set` définis dans les questions 8 et 9. La plupart des candidats s'y sont tenus mais certains ont continué à manipuler les instances de `presolution` directement.
La question a été globalement bien traitée, même si on commence à trouver du code faux comme présenté précédemment (mélange impératif/fonctionnel).
Q12 - il s'agissait de la première question de programmation qui demandait un peu de rigueur. On trouve souvent des réponses très compliquées (et fausses) alors qu'une version récursive de la fonction pouvait s'écrire de façon assez concise.
Q13 - question globalement bien traitée, même si des solutions sont parfois trop compliquées. Un nombre non négligeable de candidats a oublié de vérifier que la ligne ou la colonne était complète avant de la comparer à la trace correspondante.

Des candidats redéfinissent également l'égalité entre listes, ce qui n'était pas nécessaire (mais non pénalisant).
Q14 - question globalement bien traitée.
Q15 - cette question faisait appel à la technique de retour sur trace qui est au programme. Certains candidats « descendent » trop et cherchent à explorer l'arbre des solutions sur deux niveaux au lieu d'un, ce qui complique inutilement le code. D'autres essayent d'explorer les deux branches possibles « en même temps » au travers d'un `let` demandant la construction des deux extensions possibles (`N` et `B`).
Enfin, certains candidats oublient que le type de retour demandé est `presolution option` et non pas `presolution` (ce qui est impossible si on veut pouvoir traiter les cas où il n'existe pas d'extension complète).
Q16 - cette question débutait la partie 3 et a été globalement mal traitée. On trouve beaucoup de réponses très surprenantes.
Q17 - un nombre non négligeable de candidats n'ont pas compris ce qui était attendu et ont utilisé `I` dans l'expression régulière demandée. Au niveau de la forme, on note parfois une non maîtrise de la syntaxe des expressions régulières. Enfin, l'erreur la plus couramment faite était l'utilisation de B^* au lieu de B^+ dans l'expression (il y a au moins une case blanche entre deux cases noires).
Q18 - de façon assez surprenante, certains candidats répondent correctement à cette question alors que l'expression régulière qu'ils proposent à la question précédente ne correspond pas. On trouve toutefois un grand nombre de bonnes réponses.
Q19 - question globalement très mal traitée. On trouve très peu de bonnes réponses, alors qu'il s'agit quasiment d'une question de cours.
Q20 - le déploiement de l'automate était un peu fastidieux à écrire, surtout si on ne prenait pas le temps de remarquer qu'il fallait le faire « par étage », ce qui rendait parfois la construction de l'automate compliquée. Certains candidats ne marquent pas les états accessibles ou co-accessibles ou le font mal (en particulier pour les états co-accessibles) alors qu'ils ont dessiné le bon automate.
Q21 - la fonction à écrire pouvait être compliquée si l'on n'avait pas compris qu'il fallait propager l'information d'accessibilité par strates, comme pour le dessin de l'automate de la question précédente. La question a été globalement peu abordée.
Q22 - question très peu traitée.
Q23 - question très peu traitée.
Q24 - question très peu traitée. On trouve parfois des tentatives de démonstration qui sont incorrectes.
Q25 - question très peu traitée.
Q26 - question très peu traitée.
Q27 - question très peu traitée.

RETOUR