Concours Centrale-Supélec, France

Session 2026, filière MP

Option informatique

Durée : 4 heures

Ce que traite le sujet

- Partie A, exercice : coloration syntaxique
- Partie B, I : cas de la 2-COLORATION
- Partie B, II : cas de la 3-COLORATION
- Partie B, III : graphes planaires

Sujet reproduit sans retouche à partir de la page suivante.

Résumé des parties, chapitres de cours liés et autres années :
prepahelp.com/annales/centrale/mp/2026/info

Prepa Help ne revendique aucun droit sur le sujet, propriété de l'organisateur du concours.

CONCOURS CENTRALE•SUPÉLEC

Option Informatique

MP

2026

4 heures — Calculatrice autorisée

Coloriages

Généralités

Le sujet comporte un exercice (Partie A) et un problème (Partie B) totalement indépendants. Bien qu'il y ait une progression logique dans l'ordre des différentes sous-parties du problème, celles-ci sont largement indépendantes.

Toute fonction demandée dans une partie peut être utilisée dans la suite du sujet même si la question concernée n'a pas été traitée par le candidat.

Les questions de programmation sont à traiter exclusivement en langage OCaml. Les fonctions classiques des modules *Array* et *List* sont autorisées. Une partie de la documentation de ces modules est donnée en fin d'énoncé. Toutes les fonctions de la bibliothèque standard comme par exemple *max* ou *incr* sont autorisées.

Partie A – Exercice : Coloration syntaxique

On s'intéresse ici à l'étape d'analyse lexicale d'un code source en OCaml dans le but de réaliser une coloration syntaxique c'est-à-dire d'identifier les mots clés du langage, les variables, etc. Il s'agit de la première étape lors de la compilation et elle peut être faite en utilisant un automate. On notera que la vérification de la syntaxe ne rentre pas dans le cadre de cette analyse. On définit un type **token** (non exhaustif ici) qui permet de travailler avec les différents lexèmes reconnus.

```
type token = | Let | In | Var of string | Int of int | Lpar | Rpar | Add | Equal
```

On se limite ainsi à reconnaître et manipuler les mots clés **let**, **in**, un **nom de variable** (on se limite à une suite de caractères en minuscule autre que les mots clés), un **entier** (0 ou une suite de chiffres démarrant par un autre chiffre que 0), les **parenthèses ouvrantes** et **fermantes** et les **symboles** $'+'$ **et** $' ='$. D'autres symboles sont à oublier comme $' '$, $'\backslash n'$ ou $'\backslash t'$ car ils ne servent qu'à séparer les lexèmes.

Q1. On travaille sur l'alphabet $\Sigma = \{0, 1\}$. Proposer une expression régulière dénotant les écritures en binaire des entiers (0 ou toute suite de 0 et de 1 commençant par 1).

Q2. Proposer alors un automate fini déterministe qui reconnaît ce langage. Comment pourrait-on le modifier pour qu'il reconnaisse les écritures en base 10 des entiers ?

On donne l'automate fini non déterministe A_l suivant :

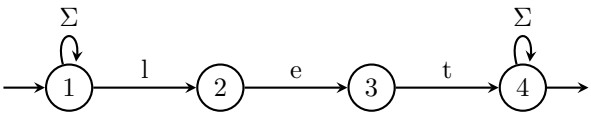


Figure 1 – Automate fini non déterministe A_l

Q3. Indiquer le langage reconnu par l'automate de la figure 1. Proposer un automate fini déterministe qui reconnaît le même langage.

Q4. Justifier que l'on peut construire un automate fini déterministe permettant de reconnaître l'ensemble des lexèmes proposés dans le type **token**.

On attribue alors un constructeur de **token** à chaque état final de cet automate en fonction de ce qu'il est censé identifier. Les constructeurs de token sont choisis de manière à ce que, si deux lexèmes correspondent à la même suite de caractères, on associe le token arrivant en premier dans l'ordre de définition des tokens (un mot clé avant un nom de variable).

En pratique, la phase d'analyse lexicale lit la liste de caractères du code considéré, en maintenant la liste des caractères lus et celle des caractères restants à lire, et en répétant le mécanisme ci-dessous.

- Si le premier caractère est $' '$, $'\backslash n'$ ou $'\backslash t'$, on l'ignore et on recommence à partir du caractère suivant.
- Sinon, on lit la liste de caractères à travers l'automate en partant de son unique état initial pour identifier un lexème : un lexème est reconnu dès que l'automate échoue lors de la lecture d'un caractère à partir d'un état acceptant ou si l'on finit sur un état acceptant (on repère ainsi le plus long lexème possible). Une fois un lexème identifié, on construit un objet de type token associé et on le stocke.
- On recommence à partir de l'éventuel caractère qui a provoqué l'échec.

On obtient à la fin la liste des tokens identifiés.

Par exemple sur la liste de caractères suivante :

```
['l'; 'e'; 't'; ' '; 'v'; 'a'; 'l'; 'e'; 'u'; 'r'; ' '; '='; ' '; '4'; '2']
```

- on reconnaît d'abord le token **Let** en échouant sur l'espace ;
- on recommence avec la chaîne qui reste ;
- on oublie le caractère espace, on recommence avec la chaîne qui reste ;
- on reconnaît le token **Var "valeur"** en échouant sur l'espace ;
- et ainsi de suite.

On obtient donc à la fin la liste de tokens suivante :

```
[Let; Var "valeur"; Equal; Int 42]
```

Q5. Proposer un automate fini déterministe, en indiquant le constructeur associé à chaque état final, permettant de reconnaître les mots clés "in", "let" et les noms de variables. On se limitera à l'alphabet $\Gamma = \{a,...,z\}$.

Q6. Quelle est la complexité globale de cette phase d'analyse lexicale qui prend une liste de n caractères et renvoie la liste des tokens identifiés ?

On note $L = \{x \in \Sigma^*, |x|_(= |x|_)\}$ où $|x|_($ (respectivement $|x|_)$) représente le nombre de parenthèses ouvrantes (respectivement fermantes) dans le mot x.

Q7. Montrer que le langage L n'est pas un langage régulier.

Q8. Comment pourrait-on identifier en temps linéaire si la liste de tokens obtenue correspond à un code bien parenthésé (indépendamment de toute autre erreur de syntaxe) ? Par exemple, `[ Int 3 ; Lpar ; Equal ; Rpar ]` est bien parenthésé alors que `[ Lpar ; Int 2 ; Add ; Int 3 ; Rpar ; Rpar ; Lpar ]` ne l'est pas. On ne demande pas de code OCaml.

Partie B – Problème : Coloration de graphes

Définitions et notations

Pour un ensemble fini E, on note $|E|$ le **cardinal** de E. On définit un **graphe non orienté** comme un couple $G = (S, A)$ où S est un ensemble fini d'éléments appelés sommets et A une partie de $\mathcal{P}_2(S)$ (ensemble des parties de S à 2 éléments) dont les éléments sont appelés **arêtes** (on considère que le graphe est sans boucle, c'est à dire que les sommets qui composent une arête sont distincts).

Un graphe non orienté $G = (S, A)$ est dit **connexe** si pour tout couple de sommets $(u, v) \in S^2$ il existe un chemin reliant u à v c'est-à-dire un entier n et des sommets $s_0,s_1,\ldots,s_n$ tels que $s_0 = u$, $s_n = v$ et pour tout $i \in [\![0,n-1]\!]$, $\{s_i,s_{i+1}\} \in A$ (n est alors la longueur du chemin).

Le **degré d'un sommet** $s \in S$ noté $deg(s)$ est le nombre d'arêtes contenant ce sommet.

On dit que deux sommets u et v sont **adjacents** si $\{u, v\} \in A$. Enfin, un cycle est un chemin de longueur supérieure ou égale à 3 reliant un sommet à lui-même dont les sommets sont tous différents (à part les extrémités).

Le graphe non orienté $G_1 = \Big(\{0,1,2,3,4,5\}, \big\{\{0,2\},\{0,3\},\{0,4\},\{1,2\},\{1,3\},\{(2,3\},\{2,5\},\{3,4\}\big\}\Big)$ est représenté sur la figure 2 ci-dessous et c'est un graphe connexe.

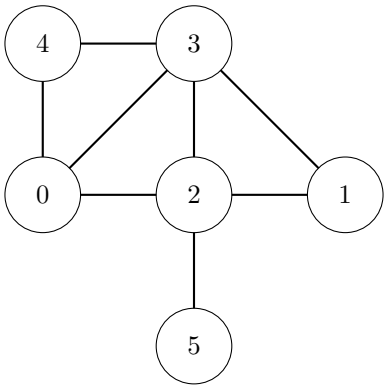


Figure 2 – Le graphe G_1

On représentera ces graphes en OCaml par le type graphe suivant :

```
type graphe = {nbr : int ; adj : int list array}
```

Pour un graphe g, $g.nbr$ désigne son nombre de sommets (qu'on suppose étiquetés par les entiers de 0 à $|S|-1$) et $g.adj$ est son tableau de listes d'adjacence ($g.adj.(i)$ désigne la liste des sommets adjacents au sommet i, dans n'importe quel ordre). Le graphes G_1 est représenté en mémoire par :

```
let g1 = {nbr = 6 ; adj = [|[3; 2; 4];
                            [2; 3];
                            [3; 0; 1; 5];
                            [0; 4; 2; 1];
                            [0; 3];
                            [2]|]};;
```

Une **coloration de sommets** d'un graphe $G = (S, A)$ est une application $c : S \to \mathbb{N}$. Elle est dite valide si $\forall \{u,v\} \in A$, $c(u) \neq c(v)$.

On parle de **k-coloration** pour une coloration à valeurs dans $[\![0, k-1]\!]$ et on dit qu'un graphe est k-colorable s'il admet une k-coloration valide.

On définit le problème de décision **k-COLORATION** de coloration avec k couleurs comme suit :

Problème : k-COLORATION

Entrée : Un graphe non orienté $G = (S, A)$

Sortie : Existe-t-il une k-coloration valide du graphe G ?

On définit le type suivant pour les colorations :

```
type col = int array
```

Pour une coloration c, $c.(i)$ indique la couleur associée au sommet i.

I – Cas de la 2-COLORATION

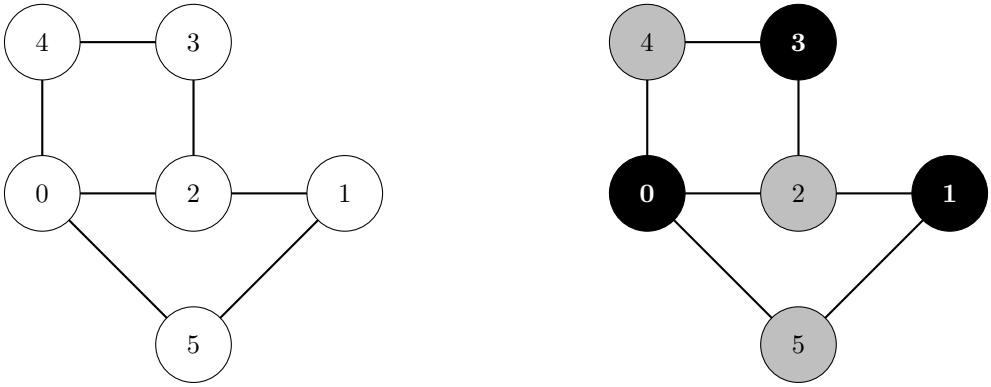


Figure 3 – Le graphe G_2 et une bicoloration valide associée

Le graphe G_2 de la figure 3 est 2-colorable mais le graphe G_1 de la Figure 2 ne l'est pas.

I.1 – graphes bipartis

Q9. Justifier qu'un graphe est 2-colorable si et seulement s'il est biparti.

Q10. Montrer qu'un graphe n'est pas 2-colorable lorsqu'il comporte au moins un cycle impair (c'est à dire composé d'un nombre impair d'arêtes).

Q11. Inversement, montrer qu'un graphe qui ne contient pas de cycle impair est 2-colorable. On pourra raisonner par récurrence sur le nombre d'arêtes.

I.2 – Une solution naïve

Nous allons considérer une 2-coloration (ou bicoloration) comme un entier décomposé en base 2 : à chaque sommet est associée la couleur 0 ou 1 et le tableau de la coloration donne les bits de l'entier correspondant, le sommet 0 correspondant au bit de poids faible. Par exemple, la coloration de la figure 3 donne le tableau $[|1;1;0;1;0;0|]$ associé à l'entier $1+2+8=11$. On va utiliser cette interprétation afin d'itérer les différentes colorations possibles.

Q12. Proposer une fonction **suivante : col -> bool** qui renvoie *true* s'il existe une bicoloration suivante (une bicoloration associée à l'entier immédiatement supérieur à celui de la bicoloration donnée), $false$ sinon et met en même temps à jour la bicoloration reçue à la bicoloration suivante. La fonction devra s'exécuter dans le meilleur cas en temps constant.

Q13. Proposer une fonction **valide : graphe -> col -> bool** qui décide si la bicoloration passée en argument est une bicoloration valide pour le graphe passé en argument.

Q14. En déduire une fonction **bicolnaif : graphe -> bool * col** qui détermine, par recherche exhaustive, s'il existe une bicoloration valide pour un graphe donné et renvoie le cas échéant une coloration adéquate.

Q15. Indiquer, en justifiant, quelle est la complexité dans le pire cas d'une telle fonction.

I.3 – un détour par les structures de données

L'objectif est de proposer une implémentation de structure de file (d'attente). L'utilisation des modules Queue et Stack est donc interdite dans toute cette partie.

On pose

```
type 'a pile = 'a list
```

Q16. En quoi la structure de liste d'OCaml est-elle une bonne implémentation de pile ?

On choisit d'implémenter une structure de file en utilisant deux piles. On définit le type file associé :

```
type 'a file = { entree : 'a pile ; sortie :  'a pile }
```

Chaque élément ajouté à la structure est empilé sur le dessus de la pile "entrée".

Pour sortir un élément de la structure, on utilise l'algorithme suivant :

```
Si la pile "sortie" est vide ALORS
    TANT QUE la pile "entree" est non vide
        dépiler l'élément en tête de pile "entree" et l'empiler sur la pile "sortie".
Dépiler et renvoyer l'élément en tête de pile "sortie"
```

Q17. Proposer les fonctions :

a. **initFile : unit -> 'a file** qui renvoie une file vide,

b. **estVideFile : 'a file -> bool** qui dit si la file reçue est vide,

c. **enfile : 'a * 'a file -> 'a file** qui ajoute un élément dans la file et renvoie la nouvelle file,

d. **defile : 'a file -> 'a * 'a file** qui sort un élément de la file et le renvoie avec la nouvelle file.

Q18. Montrer que l'ordre de sortie des éléments dans une structure de ce type est le même que celui d'entrée.

Q19. Quelle est la complexité temporelle dans le pire cas d'une opération **defile** ?

Q20. On considère un élément qui transite par une telle structure de file (c'est-à-dire y rentre puis éventuellement en sort). Encadrer le nombre d'opérations empile et dépile réalisées sur cet élément.

Q21. En déduire un encadrement du nombre d'opérations empile et dépile réalisées lors de n opérations **enfile** ou **defile** réalisées sur une file initialement vide. Que peut-on dire sur la complexité temporelle amortie des fonctions **enfile** et **defile** ?

I.4 – Une solution plus efficace

Q22. Proposer une fonction **bicol : graphe -> bool * col** qui détermine, par un parcours en largeur du graphe, s'il existe une bicoloration valide et le cas échéant en renvoie une. On supposera sans le vérifier que le graphe passé en argument est connexe.

Q23. Justifier la terminaison de la fonction **bicol**.

Q24. Indiquer, en justifiant, quelle est la complexité temporelle de la fonction **bicol**.

Q25. Justifier la correction de cette fonction.

Q26. Proposer une stratégie afin d'adapter la fonction **bicol** au cas où le graphe n'est pas connexe.

Q27. Comment pourrait-on implémenter la fonction **bicol** de la **Q22** en utilisant un parcours en profondeur ?

II – Cas de la 3-COLORATION

Il s'agit d'un problème difficile **c'est-à-dire que l'on ne sait pas résoudre en temps polynomial dans le cas général**. Nous allons donc envisager une stratégie par retour sur trace (**backtracking**). Pour cela, on initialise une coloration à la valeur -1 pour chaque sommet. La valeur -1 correspond à un sommet pour lequel aucune couleur n'a encore été attribuée. La valeur -1 est donc compatible localement avec toutes les autres couleurs.

Q28. Proposer une fonction de validation partielle **partial : graphe -> col -> int -> bool** qui décide si la coloration passée en argument est localement valide entre le sommet (de type int) passé en argument (et supposé vraiment coloré) et ses sommets adjacents.

Q29. Proposer une fonction **backtracking : graphe -> bool * col** qui détermine s'il existe une 3-coloration valide et le cas échéant en renvoie une pour le graphe (supposé connexe) passé en argument.

Q30. Indiquer, en justifiant, la complexité temporelle d'une telle stratégie dans le pire cas.

III – Graphes planaires

III.1 – Formule d'Euler pour un graphe connexe

Un **graphe planaire** est un graphe qui a la particularité de pouvoir se représenter dans un plan sans qu'aucune arête n'en coupe une autre (à part éventuellement aux sommets). Cela nécessite souvent de placer judicieusement les sommets dans le plan.

Lorsqu'un graphe planaire est dessiné sans arêtes qui se croisent, il divise le plan en un ensemble de régions appelées **faces**. Elles correspondent aux composantes connexes par arcs (au sens de la topologie usuelle) du plan privé du tracé des arêtes. En particulier, il y a toujours une face non bornée.

Pour un graphe connexe planaire, la frontière d'une face est le sous-graphe contenant toutes les arêtes délimitant cette face, et un parcours de frontière est un parcours fermé (c'est-à-dire qui part d'un sommet de la frontière et y revient) contenant toutes ces arêtes.

Le degré d'une face est la longueur minimale d'un parcours de frontière.

Par exemple, dans la figure 4 ci-dessous, le graphe de gauche possède trois faces :

- la frontière de sa face 2 se parcourt par le chemin $d - e - f - d$ donc cette face a un degré de 3,
- la frontière de sa face 3 (la face non bornée) contient les arêtes ab, ac, bc, cd, de, ef, fd et se parcourt par $a - b - c - d - e - f - d - c - a$ donc la face 3 a un degré de 8.

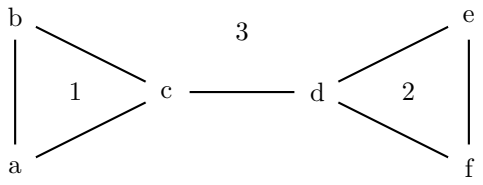


Figure 4 – Graphe planaire et faces

Dans toute cette partie, tout dessin visant à illustrer l'argumentation sera valorisé.

Q31. Soit G un graphe connexe planaire. Calculer en fonction de $|A|$ la somme des degrés de ses sommets et la somme des degrés de ses faces.

La formule d'Euler pour les graphes planaires connexes (non vides) peut s'écrire $|S| - |A| + |F| = 2$ où $|F|$ représente le nombre de faces.

Q32. Montrer en détail cette formule dans le cas des arbres.

Q33. Montrer la formule d'Euler dans le cas d'un graphe planaire connexe.

Q34. Montrer que dans le cas d'un graphe connexe planaire pour lequel $|S| \geq 3$, on a $|A| \leq 3 \times |S| - 6$ (on pourra considérer la somme des degrés de ses faces). En déduire que le graphe complet à 5 sommets (noté K_5) n'est pas planaire.

Q35. En déduire qu'un graphe connexe planaire contient forcément un sommet de degré inférieur ou égal à 5.

Q36. Montrer que dans le cas d'un graphe connexe planaire biparti pour lequel $|S| \geq 3$, on a $|A| \leq 2 \times |S| - 4$. En déduire que le graphe biparti complet ayant 3 sommets dans chaque ensemble (noté $K_{3,3}$) n'est pas planaire.

On admet le **théorème de Kuratowski** qui dit qu'un graphe fini est planaire si et seulement s'il ne contient pas de sous-graphe qui s'identifie à K_5 ou à $K_{3,3}$.

III.2 – Cas du problème Planar-2-COL

On définit le problème de décision **Planar-k-COL** par :

Problème : Planar-k-COL

Entrée : Un graphe planaire non orienté $G = (S, A)$

Sortie : Existe-t-il une k-coloration valide du graphe G ?

Nous allons voir que la difficulté de résolution de ce problème dépend fortement de la valeur de k.

Q37. Proposer une stratégie de Résolution de Planar-2-COL en temps linéaire par rapport au nombre de sommets.

III.3 – Cas du problème Planar-3-COL

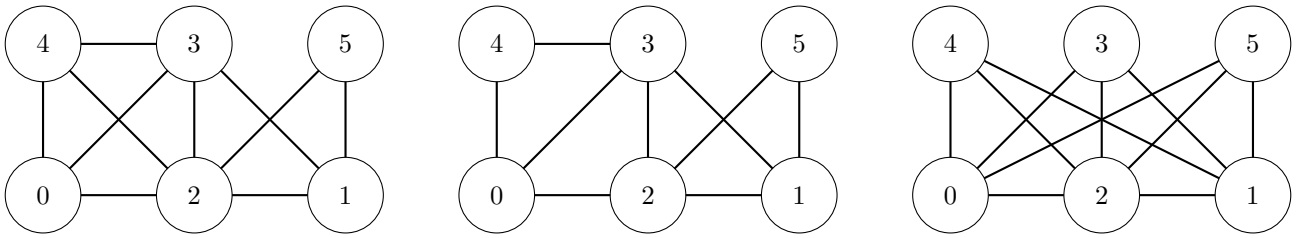


Figure 5 – De gauche à droite, les graphes G_3, G_4 et G_5

Q38. Identifier les graphes de la figure 5 qui sont planaires et 3-colorables. On justifiera ses réponses.

Q39. Proposer un graphe connexe, planaire, à 4 sommets 2-colorable. Même question afin d'obtenir un graphe connexe planaire 3-colorable mais pas 2-colorable et enfin un dernier connexe planaire 4-colorable mais pas 2 ou 3-colorable.

On peut montrer que le problème **Planar-3-COL** est aussi difficile que le problème **3-COLORATION** car pour tout graphe dont on cherche l'existence d'une 3-coloration, on peut construire un graphe en temps polynomial pour lequel résoudre **Planar-3-COL** donne la même réponse que résoudre **3-COLORATION** sur le graphe de départ.

III.4 – Cas $k \geqslant 4$

On veut montrer qu'un graphe planaire est toujours 5-colorable. On va pour cela raisonner par récurrence sur le nombre n de sommets. Les cas $n \leqslant 5$ sont élémentaires. On suppose donc la propriété vraie pour une valeur $n \geqslant 5$ et on considère un graphe planaire G ayant $n+1$ sommets. En vertu de **Q35**, on sait qu'il existe un sommet v de degré inférieur ou égal à 5.

Q40. Traiter le cas où $deg(v) \leq 4$.

On se place désormais dans le cas où $deg(v) = 5$. Le graphe étant planaire, on peut énumérer ses 5 voisins s_1,s_2,…,s_5 autour de lui en "tournant" dans le sens trigonométrique. Soit G' le sous-graphe de G obtenu en enlevant v et ses arêtes incidentes. Par hypothèse de récurrence, G' est 5-colorable. Si les couleurs associées à $s_1,\ldots,s_5$ ne sont pas toutes différentes, on conclut facilement. On suppose donc que les cinq couleurs sont utilisées et on note respectivement a,b,c,d,e les couleurs utilisées pour $s_1,\ldots,s_5$. On considère alors l'ensemble E formé

- du sommet s_1,
- des voisins de s_1 colorés avec la couleur c,
- des voisins des voisins de s_1 colorés avec la couleur a,
- des voisins des voisins des voisins de S_1 colorés avec la couleur c,
- et ainsi de suite.

Q41. Montrer que si s_3 n'appartient pas à E, on peut conclure en échangeant des couleurs.

Q42. Justifier que dans le cas contraire, il existe un cycle dans G utilisant les arêtes $\{v,s_1\}$ et $\{v,s_3\}$. s_2 et s_4 peuvent-ils être tous les deux à l'intérieur de la zone délimitée par ce cycle ou tous les deux à l'extérieur ? On ne demande pas de preuve rigoureuse mais des dessins.

Q43. Conclure.

Le théorème des quatre couleurs pour les graphes indique que tout graphe planaire est 4-colorable. Le résultat fut conjecturé en 1840 par Ferdinand Möbius. Deux premières démonstrations furent publiées, respectivement par Alfred Kempe en 1879 et Peter Guthrie Tait en 1880. Mais elles se révélèrent erronées. Les erreurs ont été relevées seulement en 1890 par Percy Heawood et en 1891 par Julius Petersen.

Finalement, en 1976, deux Américains, Kenneth Appel et Wolfgang Haken, affirment avoir démontré le théorème des quatre couleurs. Pour la première fois, en effet, la démonstration exige l'usage de l'ordinateur pour étudier les 1 478 cas critiques (plus de 1 200 heures de calcul).

ANNEXES

On donne un extrait de la documentation du langage OCaml pour le module List :

```
val length : 'a list -> int
List.length l returns the length (number of elements) of the given list l.

val hd : 'a list -> 'a
List.hd l returns the first element of the given list l.
Raises Failure if the list is empty.

val tl : 'a list -> 'a list
List.tl l returns the given list l without its first element.
Raises Failure if the list is empty.

val rev : 'a list -> 'a list
List reversal.

val iter : ('a -> unit) -> 'a list -> unit
List.iter f [a1; ...; an] applies function f in turn to [a1; ...; an].
It is equivalent to f a1; f a2; ...; f an.

val iteri : (int -> 'a -> unit) -> 'a list -> unit
Same as List.iter, but the function is applied to the index of the element
as first argument (counting from 0), and the element itself as second argument.

val map : ('a -> 'b) -> 'a list -> 'b list
List.map f [a1; ...; an] applies function f to a1, ..., an,
and builds the list [f a1; ...; f an] with the results returned by f.

val mapi : (int -> 'a -> 'b) -> 'a list -> 'b list
Same as List.map, but the function is applied to the index of the element
as first argument (counting from 0), and the element itself as second argument.

val fold_left : ('acc -> 'a -> 'acc) -> 'acc -> 'a list -> 'acc
List.fold_left f init [b1; ...; bn] is f (... (f (f init b1) b2) ...) bn.

val mem : 'a -> 'a list -> bool
List.mem a set is true if and only if a is equal to an element of set.

val filter : ('a -> bool) -> 'a list -> 'a list
List.filter f l returns all the elements of the list l that satisfy the predicate f.
The order of the elements in the input list is preserved.
```

On donne un extrait de la documentation du langage OCaml pour le module Array :

```
val length : 'a array -> int
Array.length a returns the length (number of elements) of the given array a.

val get : 'a array -> int -> 'a
Array.get a n returns the element number n of array a. The first element has number 0.
```

```
The last element has number length a - 1. You can also write a.(n) instead of get a n.
Raises Invalid_argument if n is outside the range 0 to (length a - 1).

val set : 'a array -> int -> 'a -> unit
Array.set a n x modifies array a in place, replacing element number n with x.
You can also write a.(n) <- x instead of set a n x.
Raises Invalid_argument if n is outside the range 0 to length a - 1.

val make : int -> 'a -> 'a array
Array.make n x returns a fresh array of length n, initialized with x.
All the elements of this new array are initially physically equal to x
 (in the sense of the == predicate). Consequently, if x is mutable, it is
shared among all elements of the array, and modifying x through one
of the array entries will modify all other entries at the same time.
Raises Invalid_argument if n < 0

val init : int -> (int -> 'a) -> 'a array
Array.init n f returns a fresh array of length n, with element number i initialized
to the result of f i. In other terms, Array.init n f tabulates the results of f applied
in order to the integers 0 to n-1.
Raises Invalid_argument if n < 0

val make_matrix : int -> int -> 'a -> 'a array array
Array.make_matrix dimx dimy e returns a two-dimensional array (an array of arrays)
with first dimension dimx and second dimension dimy.
All the elements of this new matrix are initially physically equal to e.
The element (x,y) of a matrix m is accessed with the notation m.(x).(y).
Raises Invalid_argument if dimx or dimy is negative

val to_list : 'a array -> 'a list
Array.to_list a returns the list of all the elements of a.

val of_list : 'a list -> 'a array
Array.of_list l returns a fresh array containing the elements of l.

val iter : ('a -> unit) -> 'a array -> unit
Array.iter f a applies function f in turn to all the elements of a.
It is equivalent to f a.(0); f a.(1); ...; f a.(length a - 1); ().

val iteri : (int -> 'a -> unit) -> 'a array -> unit
Same as Array.iter, but the function is applied to the index of the element as first argument,
and the element itself as second argument.

val map : ('a -> 'b) -> 'a array -> 'b array
Array.map f a applies function f to all the elements of a,
and builds an array with the results returned by f: [| f a.(0); f a.(1); ...; f a.(length a - 1) |].

val mapi : (int -> 'a -> 'b) -> 'a array -> 'b array
Same as Array.map, but the function is applied to the index of the element as first argument,
and the element itself as second argument.

val fold_left : ('acc -> 'a -> 'acc) -> 'acc -> 'a array -> 'acc
Array.fold_left f init a computes f (... (f (f init a.(0)) a.(1)) ...) a.(n-1),
where n is the length of the array a.

val mem : 'a -> 'a array -> bool
Array.mem a set is true if and only if a is structurally equal to an element of set
(i.e. there is an x in set such that compare a x = 0).
```

Travailler les chapitres de ce sujet

Prepa Help suit le programme de MP chapitre par chapitre : cours détaillé, méthodes, pièges, exercices et problèmes corrigés pas à pas, tirés des concours. L'essai gratuit donne accès aux chapitres offerts de ta filière, sans carte bancaire.

Chapitres liés à ce sujet

- Langage OCaml : expressions, types, filtrage, récursivité
- Complexité des algorithmes itératifs et récursifs
- Récursivité
- Architecture de l'ordinateur et représentation des nombres
- Logique, ensembles et applications

Ouvrir l'essai gratuit

Scanne le code ou clique sur le bouton.

prepahelp.com/#essai

Une question sur le concours ou sur ta filière : **écrire sur WhatsApp**

Les autres années et épreuves : prepahelp.com/annales/centrale/mp