ÉCOLE POLYTECHNIQUE **FILIÈRE MP**

CONCOURS D'ADMISSION 2001

COMPOSITION D'INFORMATIQUE

(Durée : 4 heures)

L'utilisation des calculatrices n'est pas autorisée

⋆ ⋆ ⋆

AVERTISSEMENT. On attachera une grande importance à la clarté, à la précision et à la concision de la rédaction.

On définit les *arbres* (binaires complets) de la façon suivante :

– Une *feuille* est un arbre.

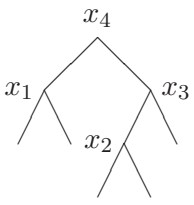
– Si g et d sont deux arbres, le couple (g, d) est un arbre.

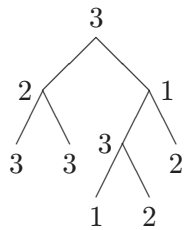
Un arbre de la forme (g, d) est un *nœud interne*, dont g est le *fils gauche* et d le *fils droit*. Par la suite, on note A_n l'ensemble des arbres possédant n feuilles ($n \geq 1$). Les feuilles d'un arbre de A_n sont numérotées de 1 à n dans l'ordre d'un parcours postfixe (dit aussi en profondeur d'abord) et de gauche à droite.

Un *flot* f de taille n est une suite de n entiers égaux à 1, 2, ou 3. Pour un arbre a de A_n fixé, cette suite définit une fonction des sous-arbres de a dans les entiers, également notée f. La fonction f associe le i-ème entier du flot f à la i-ème feuille de l'arbre a, et elle s'étend récursivement aux nœuds internes, par la formule :

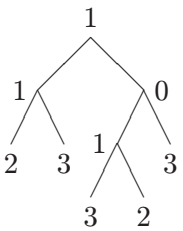
$$f((g, d)) = f(g) + f(d) \quad \text{(modulo 4)} ,$$

c'est-à-dire que si $a = (g, d)$, alors $f(a)$ est le reste de la division euclidienne par 4 de la somme $f(g) + f(d)$. On note F_n l'ensemble des flots de taille n. Un flot f de F_n est dit *compatible* avec un arbre a de A_n si, pour tout nœud interne x de a, on a : $f(x) \neq 0$. La figure suivante représente un arbre a à 5 feuilles (dont les nœuds internes sont désignés par $x_1, \ldots, x_4$), et deux flots. On notera que le premier flot est compatible avec a, tandis que le second ne l'est pas, car $f_2(x_3) = 0$.





$f_1 = [3, 3, 1, 2, 2]$



$f_2 = [2, 3, 3, 2, 3]$

Les arbres sont représentés en machine par la structure de donnée usuelle :

(* *Caml* *)

```
type arbre =
| Feuille
| Interne of arbre * arbre
```

{ *Pascal* }

```
type
  arbre = ^cellule_arbre ;
  cellule_arbre = record
    gauche, droite : arbre
  end ;
```

En Pascal, une feuille est représentée par `nil` et on pourra utiliser la fonction `noeud`, constructeur des nœuds internes :

```
function noeud (gauche:arbre ; droite:arbre) : arbre ;
var r:arbre ;
begin
  new(r) ;
  r^.gauche := gauche ; r^.droite := droite ;
  noeud := r
end ;
```

Les flots sont représentés en machine par des tableaux d'entiers.

(* *Caml* *)

```
type flot = int vect
```

{ *Pascal* }

```
const NMAX=... ;
type flot = array [1..NMAX] of integer ;
```

L'attention des candidats qui composent en Caml est attirée sur ce que les indices des tableaux commencent à zéro. Les candidats qui composent en Pascal pourront supposer que la constante `NMAX` est toujours plus grande que les valeurs de n utilisées en pratique.

NOTE. La plupart des fonctions demandées dans ce problème sont valides sous certaines hypothèses portant sur leurs arguments, hypothèses qui sont énoncées à chaque question. Le code écrit ne doit jamais vérifier les hypothèses portant sur les arguments, il doit au contraire indiquer clairement comment il les utilise le cas échéant.

Partie I. Vérification de la compatibilité des flots.

Question 1. Écrire une fonction `compte_feuilles` qui prend un arbre en argument et renvoie le nombre de ses feuilles.

(* *Caml* *)

```
compte_feuilles : arbre -> int
```

{ *Pascal* }

```
function compte_feuilles (a:arbre) : integer
```

Question 2. Écrire une fonction `compatible` qui prend en arguments un arbre a de A_n et un flot f de F_n, et qui décide si f est compatible avec a. Cette fonction devra arrêter d'effectuer des additions modulo 4 dès qu'un nœud interne x vérifiant $f(x) = 0$ est découvert.

(* *Caml* *)

```
compatible :
  arbre -> flot -> bool
```

{ *Pascal* }

```
function
  compatible(a:arbre ; f:flot) : boolean
```

Question 3. Dans cette question et la suivante, a désigne un arbre de A_n et f un flot de F_n.

a) Montrer que a possède $n-1$ nœuds internes.

b) On suppose que a n'est pas réduit à une feuille et on l'écrit $a = (g, d)$. Soit un flot f *compatible* avec a. Donner les valeurs possibles du couple $(f(g), f(d))$ selon les valeurs possibles de $f(a)$.

c) Soit v un entier égal à 1, 2 ou 3. Calculer $F(a, v)$, nombre de flots f *compatibles* avec a et tels que $f(a) = v$. En déduire le nombre de flots compatibles avec a.

Question 4. Pour un a et un f donnés, la fonction `compatible` de la question 2 effectue $N_+(a, f)$ additions modulo 4 (avant de trouver un nœud interne x tel que $f(x) = 0$ ou de revenir à la racine de a). Sur l'exemple du préambule, on a $N_+(a, f_1) = 4$ et $N_+(a, f_2) = 3$. Pour tout entier k, avec $1 \leq k \leq n-1$, on définit $\nu_k(a)$ comme étant le nombre de flots f *incompatibles* avec a et tels que $N_+(a, f) = k$.

a) Calculer $\nu_1(a)$. Sa valeur dépend elle de la forme de l'arbre a ?

b) Montrer qu'il existe un arbre a' possédant $n-1$ feuilles et tel que, pour tout $k \geq 2$, on a $\nu_k(a) = 2\nu_{k-1}(a')$. En déduire l'expression de $\nu_k(a)$ dans le cas général.

c) On admet que $N_+(a, f)$ est une bonne mesure de la complexité de l'appel de `compatible` sur a et f. En considérant une distribution équiprobable des flots, la complexité moyenne de cette fonction pour a fixé est définie comme

$$\sum_f \frac{1}{3^n} N_+(a, f)$$

où f parcourt l'ensemble des 3^n flots de taille n. Calculer cette complexité moyenne, ainsi que sa limite lorsque $n \to \infty$.

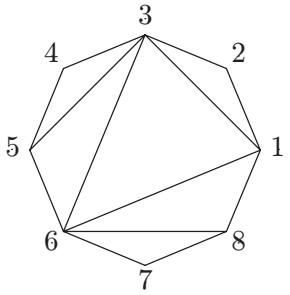
NOTE. On pourra utiliser sans la démontrer la formule suivante :

$$\sum_{k=1}^{n} k\alpha^{k-1} = \frac{1 - \alpha^n(n+1-n\alpha)}{(1-\alpha)^2}$$

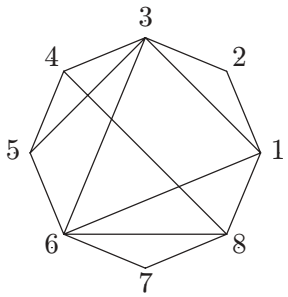
Partie II. Triangulation des polygones.

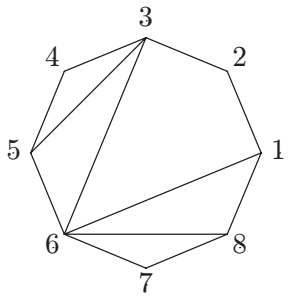
Soit P_n un polygone convexe à n côtés ($n \geq 3$), dont les sommets sont numérotés de 1 à n en suivant le sens trigonométrique. Une *corde* est un segment qui joint deux sommets non-contigus du polygone. Une *subdivision* de P_n est un ensemble de segments qui comprend au moins les côtés de P_n plus un certain nombre de cordes. Une subdivision est dite *propre* lorsque toutes ses cordes sont non-sécantes (sauf éventuellement en leurs extrémités). Une subdivision propre définit un ensemble de *faces*, qui sont les polygones (nécessairement convexes) formés à l'aide de ses segments et dont l'intérieur ne contient aucun autre segment.

Une *triangulation* de P_n est une subdivision propre dont les faces sont des triangles. Voici par exemple une triangulation de P_8 :



En revanche, les deux dessins suivants ne décrivent pas des triangulations : le premier parce que la corde qui relie les sommets 4 et 8 coupe trois autres cordes ; le second parce que la face dont les sommets sont 1, 2, 3 et 6 n'est pas un triangle.





Un segment quelconque reliant deux sommets de P_n est codé par le couple $(i, j), i < j$ de ses extrémités, et un ensemble de segments par la liste de ses éléments. On notera que les éléments d'une liste qui représente un ensemble sont supposés deux à deux distincts.

(Caml *)*

```
type segment == int * int
and segments == segment list
```

{ Pascal }

```
type
  segment = record
    debut,fin : integer
  end ;
  segments = ^cellule_segments ;
  cellule_segments = record
    tete : segment ;
    suite : segments
  end ;
```

En Pascal, on pourra utiliser le constructeur des cellules de liste `cons` :

```
function cons(tete:segment ; suite:segments) : segments ;
```

Ainsi, la triangulation donnée en exemple peut être codée par $[(1,2)\,;\,(1,3)\,;\,(1,6)\,;\,(1,8)\,;\,(2,3)\,;\,(3,4)\,;\,(3,5)\,;\,(3,6)\,;\,(4,5)\,;\,(5,6)\,;\,(6,7)\,;\,(6,8)\,;\,(7,8)]$, tandis que l'ensemble de ses cordes peut être codé par $[(1,3)\,;\,(1,6)\,;\,(3,5)\,;\,(3,6)\,;\,(6,8)]$.

Question 5. Montrer que le nombre de cordes d'une triangulation de P_n est une fonction de n et donner son expression $p(n)$.

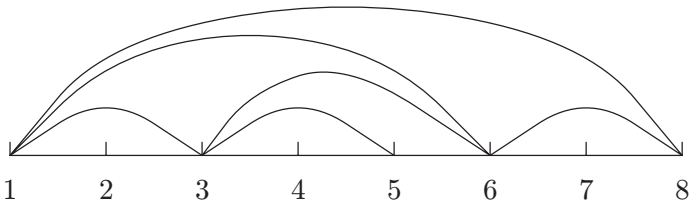
Question 6. On admet qu'un ensemble de $p(n)$ cordes non-sécantes (sauf éventuellement en leurs extrémités) définit une triangulation de P_n. Écrire une fonction `triangulation`, de

complexité $O(n^2)$, qui prend en arguments un entier n et un ensemble de cordes C de P_n, et qui décide si C définit une triangulation de P_n.

(* *Caml* *)	{ *Pascal* }
`triangulation :`	`function`
`  int -> segments -> bool`	`  triangulation(n:integer ; c:segments) : boolean`

Conformément à la note préalable, il n'appartient pas à la fonction `triangulation` de vérifier que `c` est bien une liste de segments distincts, dont chaque élément est bien une corde de P_n.

Question 7. On dessine une triangulation en représentant les $n-1$ côtés $(k, k+1)$, $1 \leq k < n$, du polygone, sous forme d'un segment de longueur $n-1$, et chaque corde (i, j), ainsi que le côté $(1, n)$, comme un arc reliant i à j. Voici le dessin représentant la triangulation donnée en exemple :



a) Utiliser cette représentation de l'exemple de triangulation pour lui associer un arbre à 7 feuilles et dont les nœuds internes et les feuilles correspondent aux segments de la triangulation.

b) Généraliser le procédé en décrivant comment on peut associer un arbre a de A_{n-1} à une triangulation t de P_n. On raisonnera en termes de polygones, il pourra être utile de remarquer que le côté $(1, n)$ est particulier.

c) Écrire une fonction `triangle_arbre`, qui prend en arguments un entier n et une triangulation t de P_n et renvoie un arbre qui représente t.

(* *Caml* *)	{ *Pascal* }
`triangle_arbre :`	`function`
`  int -> segments -> arbre`	`  triangle_arbre(n:integer ; t:segments):arbre`

Question 8. Écrire la fonction inverse de la fonction de la question précédente. C'est-à-dire, programmer la fonction `arbre_triangle` qui prend en argument un arbre a de A_{n-1} et renvoie la triangulation de P_n représentée par a.

(* *Caml* *)	{ *Pascal* }
`arbre_triangle :`	`function`
`  arbre -> segments`	`  arbre_triangle (a:arbre) : segments`

NOTE. On ne cherchera pas à prouver que les fonctions `arbre_triangle` et `triangle_arbre` sont inverses l'une de l'autre.

Partie III. Les quatre couleurs.

Le *problème des quatre couleurs* consiste à colorier une carte géographique avec au plus quatre couleurs, de telle sorte que deux pays voisins soient coloriés différemment. Le *théorème des quatre couleurs* affirme qu'une telle coloration est possible pour toute carte dessinée sur une sphère.

On admettra que la coloration d'une carte se ramène à la coloration d'une double triangulation d'un polygone P_n, qui à son tour se ramène à la construction d'un flot compatible avec deux arbres donnés de A_{n-1}. Le théorème des quatre couleurs exprime qu'un tel flot existe toujours. L'objet de cette partie est de vérifier expérimentalement ce théorème.

Étant donné un ensemble fini E, de cardinal c, une *génération* de E est une bijection de l'intervalle entier $[0 \dots c-1]$ dans E.

Question 9. Soient E et E', deux ensembles finis, générés respectivement par ϕ et ϕ'.
a) Donner une génération du produit cartésien $E \times E'$.
b) On suppose que E et E' sont disjoints. Donner une génération de l'union $E \cup E'$.
c) On suppose que E et E' sont disjoints et de même cardinal. Donner une autre génération de l'union $E \cup E'$, qui n'utilise pas la valeur du cardinal de E.

Question 10. On note $N_a(n)$ le nombre d'arbres à n feuilles ($N_a(n)$ est le cardinal de A_n).
a) Exprimer $N_a(n)$ sous forme d'une récurrence faisant intervenir les $N_a(i)$ avec $1 \leq i < n$. En déduire une fonction (procédure en Pascal) `calcule_na` qui prend un entier n en argument et range les $N_a(i)$ avec $1 \leq i \leq n$, dans un tableau global d'entiers `na` réputé de taille suffisante.

(Caml *)*	*{ Pascal }*
`calcule_na : int -> unit`	`procedure calcule_na(n:integer)`

b) Construire une génération de A_n, c'est-à-dire écrire une fonction `int_arbre` qui prend en arguments un entier n et un entier k compris entre zéro et $N_a(n)-1$ et renvoie un arbre à n feuilles.

(Caml *)*	*{ Pascal }*
`int_arbre : int -> int -> arbre`	`function int_arbre (n,k:integer) : arbre`

Question 11. Étant donnés a, un arbre à n feuilles, et v, un entier égal à 1, 2 ou 3, il est rappelé (cf. question 3) que $F(a, v)$ est le cardinal de l'ensemble des flots f compatibles avec a et tels que $f(a) = v$.

Construire une génération de cet ensemble, c'est-à-dire écrire une fonction `int_flot` qui prend en arguments un entier n, un arbre a à n feuilles, un entier v compris entre 1 et 3 et un entier k compris entre 0 et $F(a, v) - 1$, et qui renvoie un flot f de taille n tel que $f(a) = v$. En Caml on se conformera au type suivant :

```
int_flot : int -> arbre -> int -> int -> flot
```

En Pascal, le flot sera renvoyé par le truchement d'un argument supplémentaire passé par variable.

```
procedure int_flot (n:integer ; a:arbre ; v,k:integer ; var f:flot)
```

Question 12.

a) Écrire une fonction (procédure en Pascal) `trouve_compatible` qui prend en arguments un entier n et deux arbres à n feuilles a et b, et qui renvoie un flot compatible à la fois avec a et b. La terminaison de `trouve_compatible` doit être garantie.

(* *Caml* *)	*{ Pascal }*

`trouve_compatible :`
`  int -> arbre -> arbre -> flot`
	`procedure`
	`  trouve_compatible`
	`  (n:integer ; a,b:arbre ; var f:flot)`

b) Écrire une fonction (procédure en Pascal) `quatre_couleurs` qui prend en argument un entier n, tire au hasard deux arbres à n feuilles et renvoie un flot compatible avec ces deux arbres.

(* *Caml* *)	*{ Pascal }*
`quatre_couleurs : int -> flot`	`procedure` `  quatre_couleurs (n:integer ; var f:flot)`

On pourra utiliser la fonction `random_int` qui prend un entier max en argument et renvoie un entier aléatoire compris entre zéro et $max - 1$.

Rapport de M. Luc MARANGET, correcteur.

De l'épreuve

Rappelons que, si l'ensemble des candidats qui ont choisi cette option passent l'épreuve d'informatique, seules les copies des candidats admissibles sont corrigées. Cette année j'ai corrigé 206 copies (200 français et 6 étrangers) dont 22 étaient rédigées en Pascal. L'épreuve s'est révélée difficile et la notation a du être adaptée, produisant finalement une moyenne de 10,4 (10,5 pour les candidats français) et un écart type de 3,7. La répartition des notes est la suivante.

	$0 \leq N < 4$	$4 \leq N < 8$	$8 \leq N < 12$	$12 \leq N < 16$	$16 \leq N < 20$
Pascal	5 %	32 %	45 %	18 %	0 %
Caml	4 %	25 %	35 %	28 %	8 %
Total	4 %	26 %	36 %	27 %	7 %

Ces catégories ne rendent pas compte de ce que la répartition des notes n'est pas classique. L'effectif des candidats qui ont autour de 10 est relativement faible, tandis qu'on constate des regroupement de notes légèrement au dessous et au dessus de cette moyenne. L'épreuve a été nettement moins bien réussie par les candidats qui ont composé en Pascal (leur moyenne est de 9,0).

Du problème

La démonstration du théorème des quatre couleurs a marqué une date importante dans l'évolution du rapport entre l'informatique et les mathématiques. En effet, l'étape finale de cette démonstration a recours à l'énumération de nombreux cas, et un ordinateur est ici tout a fait approprié. Toutefois, il faut bien reconnaître que le problème proposé cet année n'a qu'un lointain rapport avec cette démonstration. Il s'agissait plutôt de variations sur une structure d'arbre, définie conformément au programme, non comme un cas particulier des graphes, mais comme une structure récursive. Ce point de vue a généralement été suivi. Toutefois, la définition d'une structure d'arbre qui n'exprime qu'une forme et dont les nœuds ne portent aucune information a désorienté certains candidats. Je trouve cela un peu dommage, si l'on songe que les types des arbres proposés sont les plus simples possibles.

Le problème adoptait une structure en trois parties largement indépendantes les une des autres (à l'exception des deux dernières questions de la troisième partie, qui faisaient référence à la première partie). Ce type de structure permet à certains candidats d'engranger des points en ne comprenant le problème qu'imparfaitement. Mais, s'approprier chacune des parties suffisam-

ment pour réussir les questions rénumératrices n'était pas facile. C'était en particulier le cas de la première partie et ceci dès la deuxième question, puis de la deuxième partie à partir de sa troisième question. Cette répétition de difficultés explique peut-être que certains candidats se sont trouvés déstabilisés prématurément. Pourtant ces question n'étaient pas *bloquantes* et le temps passé dessus permettait d'aborder les questions suivantes en connaissance de cause.

De l'évaluation

Cette année, je donne dans le rapport détaillé qui va suivre, les critères principaux utilisés pour la notation. Ainsi, les futurs candidats pourront, en confrontant ces remarques avec l'énoncé, s'initier à la lecture d'un énoncé d'informatique.

Dans cet esprit, je livre dès maintenant quelques critères d'ordre général. Je corrige les questions qui demandent d'écrire du code en lisant d'abord le code, si le code est juste je ne tiens pas trop compte des commentaires (qui sont parfois faux). Dans le cas contraire, je les lis attentivement pour comprendre l'idée du candidat, toutefois le candidat a déjà perdu une bonne partie de ses points, car répondre à ces questions c'est écrire un programme. Dans certains cas, un commentaire exceptionnellement pertinent sauve partiellement la situation.

En lisant les programmes, je ne m'arrête pas exagérément à certaines erreurs, *à condition de comprendre ce que le candidat a voulu écrire*. Ainsi, je ne sanctionne pratiquement jamais les `begin` ... `end` mal imbriqués, lorsque l'indentation du code révèle l'intention, ni les « `!` » manquants sur les références en Caml. En revanche, je sanctionne les `else` manquants, car il manque alors du code important, et suis plus circonspect au sujet des « `=` » pour « `:=` » surtout si ils sont précédés d'un `let`. Au fond, les qualités de rigueur et de non-ambiguïté nécessaires pour parler à une machine valent aussi pour le correcteur. Mais, s'y ajoutent la clarté et la transparence des intentions qui prennent souvent le pas sur les détails de syntaxe.

Toutefois, il faut bien produire un programme. Par conséquent, tout « code » non-effectif est sanctionné, 2^n est un exemple rebattu, ainsi que ses variations `2^n`, etc., mais cette année j'ai eu la surprise de découvrit un opérateur « Σ », parfois noté `sum`. Un autre exemple du même style est le test d'appartenance à un intervalle, parfois noté $x < y < z$ mais qui se programme par `x < y && y < z`. Les méconnaissances graves de certains principes fondamentaux du langage adopté sont sanctionnées, il s'agit souvent d'erreurs liées au typage. Par exemple, la valeur rendue par une fonction n'a pas un type bien clair, parce qu'il manque une branche `else`, ou on a écrit une boucle

dont le corps est de type booléen. En Caml encore, le pouvoir du filtrage est parfois surestimé. Je précise donc que :

1. Un motif ne peut pas contenir deux fois la même variable, ni contenir d'appel de fonction. Ainsi, le code suivant est doublement incorrect :

   ```
   let f = function
   | (i,i+1) -> true
   | _       -> false
   ```

2. Les variables contenues dans un motif sont liantes. La fonction `f` suivante ne sélectionne pas un couple de la forme (i, a) ou (a, j) avec i et j donnés :

   ```
   let rec f i j = function
   | (i,a)::_ -> a
   | (a,j)::_ -> a
   | _::rem   -> f i j rem
   | []       -> failwith "Pas possible"
   ```

   La fonction `f` renvoie en fait la deuxième composante du premier couple de la liste passée en argument. Ses deuxième et troisième cas sont redondants.

Sont également sanctionnées les violations de règles simples de bonne programmation, je n'en citerai que deux : une boucle sur les listes (mais pourquoi donc écrire une telle boucle en Caml ?) doit progresser (`p := tl !p` ou `p := p^.suite` doit se trouver quelque part) et un code qui travaille sur les listes doit régler le cas de la liste vide. Des précisions sur ce dernier point, plus subtil qu'il n'y paraît seront données au sujet de la question 6.

D'autres questions sont des démonstrations, je juge ici à la fois la structure globale de la preuve (ce qui revient finalement à juger d'abord la stratégie adoptée puis la clarté de la rédaction) et la présence de quelques arguments clés. Chaque affirmation dans une preuve appelle une justification, et les meilleures justifications font appel aux définitions de l'énoncé. Des précisions seront données au sujet de la question 5.

Dans le rapport détaillé qui suit, est indiqué, pour chaque question, le pourcentage des candidats qui ont obtenus au moins la moitié des points.

Question 1. [92 %] Ici et pour cette seule question le code doit être parfait (même la casse des lettres compte en Caml). La plupart des réponses ne rapportant aucun point traduisent une erreur d'interprétation de la définition de type de l'énoncé (représenter un nœud interne par une paire).

Question 2. [42 %] Cette question est difficile pour une deuxième question.

Il faut en effet résoudre trois difficultés de programmation, d'abord gérer correctement l'attribution des cases du flot aux feuilles, ensuite combiner deux catégories de résultats (entiers pour la valeur des $f(a)$ et booléens pour le résultat de la compatibilité), enfin interpréter et réaliser la condition de l'énoncé sur l'arrêt des calculs.

La première difficulté a bloqué bien des candidats, pourtant les solutions étaient nombreuses : partition explicite du flot, utilisation de `compte_feuilles` ou d'un compteur des feuilles vues. On peut, pour résoudre le deuxième point, avoir recours à une paire formée d'un entier et d'un booléen, mais les solutions élégantes remarquent que $f(a)$ n'est pas nul si f et a sont compatibles. Dès lors on peut écrire une fonction des arbres dans les entiers qui renvoie $f(a)$ dans le cas d'un flot compatible et zéro dans le cas d'un flot incompatible. Il reste à traduire le résultat en un booléen par un test « `<> 0` ». Le troisième point se résout par *deux* expressions conditionnelles « `if` » avant d'effectuer la somme modulo 4. Ceci, afin d'abord d'éviter les calculs dans le second fils si le premier s'est au préalable révélé incompatible, puis d'éviter de sommer si le second fils s'est révélé incompatible. Utiliser un connecteur booléen, même « paresseux » conduit inévitablement, soit à un double appel préalable (dont le second pouvait se révéler inutile), ou à un renouvellement des deux appels dans le cas compatible (ce qui mène à une complexité exponentielle très mal venue). Comme souvent, l'utilisation de constructions de programmation les plus élémentaires (liaisons, expressions « `if` », ...) mène sûrement à la solution.

Toutefois et en Caml seulement, les exceptions fournissent une autre façon de résoudre les deux derniers points d'un seul coup. Il est certain que cela doit paraître un peu magique à la plupart des candidats qui maîtrisent mal cette construction de programmation dite avancée. De façon peut-être un peu provocante, je conseillerais aux candidats d'apprendre à se passer des exceptions, plutôt que de chercher à les maîtriser. Je crois que le peu de temps dont ils disposent sera alors mieux investi.

Question 3. [73 %] Le **a)** de cette question se résout sans surprise par récurrence. Les points importants sont le choix du cas de base (les feuilles), la bonne définition de la propriété prouvée et la justification de l'application de l'hypothèse de récurrence. Car à une première preuve simple, correspond une correction un peu tatillonne. Certaines solutions font appel à l'existence d'un sous-arbre à deux feuilles, dans tout arbre à deux feuilles ou plus. L'existence de ce sous-arbre doit se justifier par la finitude des arbres. D'autres solutions, plus globales, font appel à un décompte des pères et des fils, ou à des analogies avec les expressions arithmétiques ou les tournois sportifs. Je les ai

appréciées, car elle dégagent une impression de bonne compréhension de la structure d'arbre. Je suis agréablement surpris de voir apparaître des récurrences dites « structurelles », toutefois les candidats qui raisonnent ainsi hésitent encore un peu à se s'affranchir d'un entier quelconque (*taille*, hauteur, ...) décroissant avec la structure.

Le **b)** est très facile, il constitue plutôt une indication pour la question suivante. On répond au **c)** de façon sûre par une récurrence, le critère de qualité principal étant ici la bonne explication de l'expression numérique utilisée dans la récurrence. On peut aussi procéder par dénombrement plus global des flots possibles, mais alors chaque mot compte. Il convient en particulier de bien signaler qu'à v *fixé*, deux choix sont possibles à chaque nœud interne et *aucun* aux feuilles de l'arbre.

Question 4. [39 %] Cette question regroupe deux dénombrements de flots contraints. Dans le cas du **a)**, la somme de deux valeurs du flot (dont les indices sont dépendent de l'arbre a) doit être nulle les autres valeurs sont quelconques. La plupart des candidats produisent l'expression correcte, assortie d'une justification suffisante, mais on note quand même une impressionnante variété de résultats faux ($3(n-1)$, 3, 4, ...), qui sont de pures erreurs de dénombrement.

Les candidats imprécis sont attendus au **b)**. Les critères de notation sont d'abord une définition précise de a' (désigner par exemple le nœud interne à remplacer par une feuille comme le lieu de la première addition), puis une rigueur raisonnable dans le décompte des flots (qu'aurais-je vu si la solution n'était pas donnée dans l'énoncé !).

Pour le **c)**, je regarde d'abord si la première ligne de l'expression est correcte (ne pas oublier les flots compatibles, et leur assigner le bon coût : $n-1$).

Question 5. [40 %] Il s'agit d'une question de géométrie élémentaire. La géométrie offre d'intéressantes possibilités de raisonnement conçus à l'aide de moyens élémentaires. Pour fixer les idées, je donne deux réponses possibles commentées. La première est celle des candidats qui vont au plus simple, la seconde celle des candidats qui ont des connaissances.

> Montrons que pour toute triangulation de P_n on a $p(n) = n-3$, par récurrence dite forte sur n. [*Préciser la propriété prouvée et la technique de preuve*].
>
> – Cas de base pour $n = 3$. L'unique subdivision du triangle est bien une triangulation et possède bien zéro corde [*plus élégant que « évident »*].

– Sinon ($n > 3$), une triangulation de P_n possède au moins une corde c (car une subdivision sans corde admet P_n comme unique face et $n > 3$) [*argument important, car ici intervient le fait que les faces sont des triangles*]. La corde c divise P_n en deux sous-polygones P_m et P_o, avec $m+o = n+2$ (les extrémités de c sont des sommets communs à P_m et P_o) et $3 \leq m < n$, $3 \leq o < n$ [*utile pour appliquer l'hypothèse de récurrence*]. Par ailleurs, les cordes de P_n autres que c ne coupent pas c [*intervention de la propreté*] et se répartissent [*choix crucial du mot*] donc entre P_m et P_o, qui sont triangulés [*utile pour appliquer l'hypothèse de récurrence*]. On peut donc poser par hypothèse de récurrence $p(n) = p(m)+p(o)+1$, puis conclure $p(n) = (m-3)+(o-3)+1 = n-3$.

Un autre exemple s'inspire de la relation d'Euler, il s'agit d'une relation plutôt générale sur les graphes planaires. Il vaut mieux, je crois, en démontrer un cas particulier, si on ne sait pas en préciser les conditions d'application.

> Soit une triangulation de P_n. On note F le nombre de ses faces et C le nombre de ses cordes.
>
> On a la relation $F = C + 1$. En effet, en enlevant une corde d'une subdivision propre on fusionne deux faces et, en itérant le procédé, on aboutit à une subdivision sans corde et possédant une face. [*Il s'agit d'une preuve rapide par récurrence sur le nombre de cordes des subdivisions propres, à n constant.*]
>
> D'autre part, un côté appartient à une unique face, tandis qu'une corde appartient à deux faces. [*Cela peut passer pour une conséquence de la définition des faces.*] On a donc, en comptant les côtés des faces ici triangulaires, la relation $3F = 2C + n$.
>
> Des deux relations, il vient $C = n - 3$.

Question 6. [33 %] Cette question de programmation est assez classique. Les critères de notation sont un test correct de vérification que deux cordes se croisent, une bonne organisation du programme afin d'appliquer ce test à tous les couples de cordes et enfin la vérification du nombre de cordes. Dans l'ensemble, un programme organisé selon le schéma décrit plus haut a de bonnes chances d'être juste, ou si il est partiellement faux, de récolter des points. Au sujet du test de croisement, notons que si une fonction renvoie `true` lorsque ses deux arguments ne se croisent pas, il vaut mieux l'appeler `non_secant` que `secant`. Notons encore qu'adopter des noms identiques pour les sommets en jeu à la fois dans l'explication et dans le code simplifie mon travail, mais surtout celui du candidat.

Certains candidats ont cherché à optimiser l'organisation des tests, en écrivant par exemple :

```
(* teste l'existence d'un couple de cordes s\'ecantes *)
let rec existe_secant = function
| [corde] -> false
| corde :: reste ->
   ... || existe_secant reste
```

Ou en Pascal :

```
   existe_secant := false ;
   while (not existe_secant) and (p^.suite <> nil) do begin
      existe_secant := ... ;
      p := p^.suite
   end
```

Malheureusement ces programmes échouent dans le cas d'une liste vide, et ce cas se présente effectivement même si on a vérifié la longueur de la liste au préalable (si n vaut 3). Or, en remplaçant le motif `[corde]` par `[]` et le test `p^.suite <> nil` par `p <> nil`, on obtient un code équivalent pour toutes les listes non vides, et qui traite la liste vide. Il faut être effrayé par la liste vide à bon escient, c'est à dire traquer les erreurs (échec de filtrage ou déréférencement de *nil*) et ne pas rechigner à effectuer les cas de base sur la liste vide.

Certains candidats supposent, parfois explicitement (ce qui est déjà un moindre mal), que la liste est triée selon un certain ordre. Ce n'est pas légitime, le programme demandé doit suivre les conditions de l'énoncé. Si besoin était, ces candidats auraient pu écrire une fonction de tri. Enfin, on note des oublis de la la condition $i < j$, ce qui complique encore le code.

Question 7. [15 %] Les trois sous-parties sont jugées dans le détail sur des critères différents, mais leur cohérence a également joué. Le **a)** demande simplement un arbre (ne pas oublier l'association entre les nœuds et les segments, idéalement représentée par un arbre étiqueté), le **b)** demande de décrire comment obtenir un tel arbre dans le cas général, et le **c)** est la programmation effective de la construction de l'arbre. Mais l'écriture du programme aurait souvent dû conduire les candidats à réviser leur rédaction de la question **b)**. Une copie appréciée a, par exemple, d'abord répondu insuffisamment à **b)**, puis à **b)** et à **c)** en même temps, les commentaires expliquant le code étant suffisamment abstraits pour que cette démarche, clairement assumée, soit fructueuse.

Dans le cas du **b)**, il convient bien d'associer un arbre à une triangulation, (et

non pas l'inverse, même si c'est au fond la même chose). La solution tient en deux descriptions précises : où s'arrêter (à un segment on associe une feuille), et sinon comment diviser un polygone triangulé en deux sous-polygones triangulés (distinguer un côté, en déduire une face unique). Les difficultés sont la précision de la rédaction (grandement soutenue par une figure), et la généralisation des polygones pour y inclure les segments. D'autres constructions de l'arbre, par exemple à partir des feuilles, aboutissent toutes à un « *On itère le procédé* » guère convaincant.

Le **c)** réclame, d'abord une recherche effective de la face définie ci-dessus (ou plus précisément de son troisième sommet), puis une gestion précise des numéros des sommets des polygones. On se rend alors compte que, d'une part, il y a plusieurs méthodes possibles pour trouver le troisième sommet, et que, d'autre part, il n'est pas forcément utile de renuméroter ou de diviser les ensembles de cordes. Donc, lâcher le stylo pour réfléchir permet de sélectionner la solution qui donne le programme le plus facile à écrire. Réciproquement, les choix de programmation bien justifiés éclairent parfois les concepts esquissés un peu rapidement au préalable, mais ces concepts doivent au final être clairement définis. Soit, pour noter **b)** au vu de **c)**, je me suis posé ce style de questions : En plus du code, le candidat m'explique-t-il pourquoi il sélectionne ce sommet pour subdiviser ? Désigne-t-il clairement le polygone passé moralement en argument à sa fonction ? Pour **b)** et **c)** ensemble, une erreur dans le cas de base était coûteuse.

Notons enfin pour ne pas paraître exagérément normatif qu'un candidat a répondu très brièvement et de façon acceptable à **b)** en remarquant que le dessin de la question **a)** s'obtient en « *ouvrant* » le polygone du préambule. J'aurais préféré un autre mot (déplier ?) et la remarque supplémentaire que le dessin de la question **a)** est en quelque sorte un arbre.

Question 8. [25 %] Cette question est en fait beaucoup plus facile que la précédente. Les critères de jugement sont la présence de tous les segments et leur bonne numérotation.

Question 9. [72 %] Cette question facile, appelle une réponse précise et brève, typique d'un problème d'informatique. L'énoncé demande des génération de trois ensembles, il faut donner ces générations. On ne demande certainement pas de prouver leur existence, on aura du mal à se contenter de leurs fonctions réciproques.

Question 10. [15,5 %] La formule demandée au **a)** a été trouvée par presque tous ceux qui on abordé la question (si on excepte les candidats fâchés avec

les dénombrements des produits cartésiens et des unions disjointes). Il n'en va pas de même du programme associé. La notation a surtout porté sur la capacité des candidats à traduire une sommation en programme. Il faut bien traiter le cas de $N_a(1)$, auquel la formule de sommation ne s'applique pas, ne rien laisser supposer implicitement sur les valeurs initiales des cases du tableau global (le faire explicitement, ou bien mieux initialiser soi-même), ne pas se tromper sur les noms des variables, et correctement gérer les éventuels décalages entres indices de la sommation et du programme. Une solution naturelle s'écrit en deux boucles `for` imbriquées.

De façon surprenante pour moi, certains candidats peinent à interpréter l'énoncé et écrivent une fonction qui alloue un tableau. En Caml, de nombreux candidats ne voient pas que la condition vague « *tableau réputé de taille suffisante* » de l'énoncé les autorise à ranger $N_a(i)$ à l'emplacement i du tableau, c'est à dire à ne pas utiliser `na.(0)`. Ces candidats devront par la suite ne pas oublier que `na.(i)` contient en fait $N_a(i+1)$.

Le **b)** est plus original. Il faut interpréter complètement en termes ensemblistes le dénombrement du **a)** et se servir des questions **9-b)** puis **9-a)**. C'est la programmation de cette interprétation qui a été notée (l'interprétation seule et suffisamment précise pouvant rapporter une fraction de point). La difficulté majeure est la détermination correcte de la répartition des feuilles entre fils gauche et droit de l'arbre numéro k. La plupart des candidats qui ont compris comment procéder ont recours à une boucle `while`, destinée à déterminer dans quel sous-intervalle approprié de $[0 \dots N_a(n)[$ se situe le paramètre k. Dès lors, il y a des difficultés spécifiques : la condition de boucle doit elle être une inégalité stricte ou pas ? Lorsque cette condition est remplie, est t-on allé un cran trop loin ou pas ? Pour s'en tirer il faut certainement bien remarquer que l'on recherche à invalider la condition de la boucle `while`, pour en déduire la propriété vraie en sortie de boucle. Il convient aussi d'essayer mentalement le programme dans disons deux cas simples et tangents ($k = 0$ et $k = N_a(n-1) - 1$ par exemple). Sans bannir les boucles, je dois dire qu'une solution récursive évite adroitement toutes ces difficultés.

Question 11. [7,8 %] Cette question ressemble un peu à la précédente. Elle est à mon avis plus difficile à (bien) concevoir et plus facile à programmer une fois conçue. La conception juste a donc été plus récompensée qu'à la question précédente. Plus précisément, une gestion juste du paramètre k a été largement récompensée et une gestion erronée largement pénalisée. Toutefois, il fallait aussi résoudre effectivement le problème d'associer l'un où l'autre des couples de la question **3-b)** à un v donné, les solutions qui répètent trois fois le même code étant jugées défavorablement. Pire, celles qui disent « *on*

réécrit le même code » sont pénalisées. De fait, on demande au candidat de résoudre la difficulté de programmation posée et non pas de dire comment il pourrait le faire. On notera au passage le lien étroit entre cette question et le calcul de $F(a, v)$ demandé dans la première partie, et en particulier avec la preuve de $F(a, v) = 2^{n-1}$ par dénombrement direct.

Question 12. [11,7 %] La notation du **a)**, plutôt facile, a porté sur la capacité des candidats à écrire un parcours (pour v variant entre 1 et 3 et k entre 0 et $F(a, v) - 1$) interrompu prématurément. Utiliser des boucles mène à des difficultés de programmation spécifiques (mais je me répète). Un candidat attentif a simplifié son code itératif en remarquant que la borne supérieure sur k est inutile. Quelques solutions ex-nihilo, qui énumèrent tous les flots au lieu d'utiliser la question précédente, ont apporté une fraction de point, à condition d'être essentiellement justes et codées élégamment.

Enfin, **b)** est évident et teste la compréhension globale du problème posé, mais aussi la persistance en fin d'épreuve du souci du détail dans les bornes (appliquer `random_int` à $N_a(n)$) et les paramètres (respecter les types des fonctions utilisées).