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 :

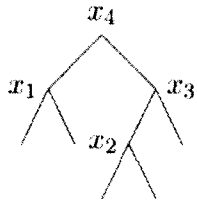
- Une *feuille* est un arbre.
- Si g et d sont deux arbres, le couple (g, d) est un arbre.

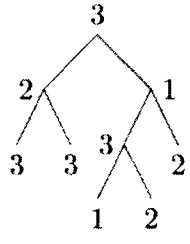
Un arbre de la forme (g, d) est un *nœud interne*, dont g est le *fils gauche* et d le *fils droit*. Par la suite, on note A_n l'ensemble des arbres possédant n feuilles ($n \geq 1$). Les feuilles d'un arbre de A_n sont numérotées de 1 à n dans l'ordre d'un parcours postfixe (dit aussi en profondeur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

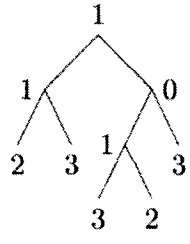
$$f((g,d)) = f(g) + f(d) \quad \text{(modulo 4)} ,$$

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 $n \to \infty$.

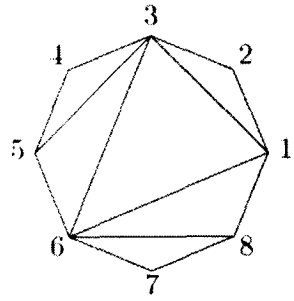
NOTE. On pourra utiliser sans la démontrer la formule suivante :

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

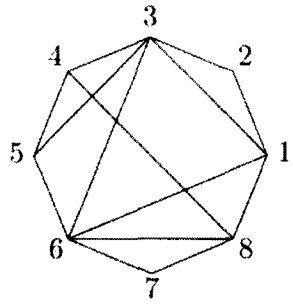
Partie II. Triangulation des polygones.

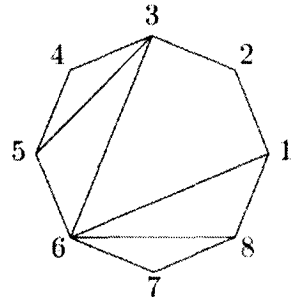
Soit P_n un polygone convexe à n côtés ($n \geq 3$), dont les sommets sont numérotés de 1 à n en suivant le sens trigonométrique. Une *corde* est un segment qui joint deux sommets non-contigus du polygone. Une *subdivision* de P_n est un ensemble de segments qui comprend au moins les côtés de P_n plus 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 $p(n)$.

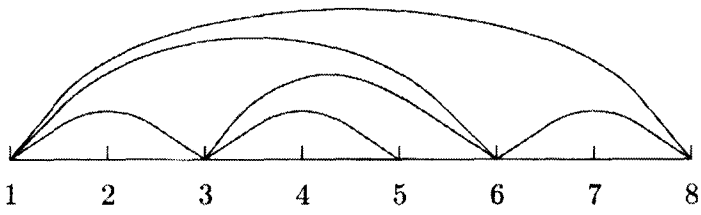
Question 6. On admet qu'un ensemble de $p(n)$ cordes non-sécantes (sauf éventuellement en leurs extrémités) définit une triangulation de P_n. Écrire une fonction `triangulation`, de

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 $[0 \ldots c-1]$ 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* *)

```
trouve_compatible :
  int -> arbre -> arbre -> flot
```

{ *Pascal* }

```
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* *)

```
quatre_couleurs : int -> flot
```

{ *Pascal* }

```
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 $max - 1$.