Option informatique — Centrale 2008

Partie I — Les mots de Lukasiewicz

Question I.1 *(Quelques propriétés)*

I.1.a Il n'y a qu'un mot de Lukasiewicz de longueur 1 : (-1), aucun de longueur 2, et un seul de longueur 3 : $(+1,-1,-1)$.

Si n est pair, comme toute lettre de l'alphabet est un entier impair, la somme $\sum_{i=1}^{n} u_i$ est nécessairement paire : il n'y a donc pas de mot de Lukasiewicz de longueur paire.

I.1.b On écrit une solution utilisant une fonction auxiliaire récursive `vérifie` dont l'argument `s` représente la somme des lettres déjà visitées.

```
let teste_Lukasiewicz u =
  let rec vérifie s = function
    | [] -> failwith "Mot vide interdit"
    | [ x ] -> s+x = -1
    | x :: v -> let s' = s+x in s'>=0 && vérifie s' v
  in
    vérifie 0 u ;;
```

Programme 1 test d'un mot de Lukasiewicz

I.1.c Notations : pour tout mot m, notons $\|m\|$ la somme de ses lettres, et appelons-la le poids de m, et notons $m[i..j]$ le facteur de m qui commence à m_i et finit à m_j, où l'on a supposé $1 \leqslant i \leqslant j \leqslant |m|$.
Notons $w = (+1)\cdot u \cdot v$, $\ell = |u|$ et $\ell' = |v|$.
Alors, avec ces notations : $\|w[1..1]\| = 1 \geqslant 0$, et, si $1 \leqslant i < \ell$, $\|w[1..i+1]\| = 1 + \|u[1..i]\| \geqslant 1 \geqslant 0$, puis $\|w[1..\ell+1]\| = 1 + \|u\| = 0$.
Ensuite, si $1 \leqslant j < \ell'$, $\|w[1..j+1+\ell]\| = 0 + \|v[1..j]\| \geqslant 0$ et enfin $\|w\| = 0 + \|v\| = -1$.
On a bien vérifié que w est encore de Lukasiewicz.

I.1.d Soit maintenant w un mot de Lukasiewicz de longueur $|w| = n \geqslant 3$. Remarquons tout d'abord que $w_1 = +1$ et que $w_n = -1$.
Supposons que w admettent deux décompositions de la forme indiquée : $w = (+1)\cdot u \cdot v = (+1)\cdot u' \cdot v'$, avec par exemple u préfixe strict de u'.
Mais u est un mot de Lukasiewicz, et donc $\|u\| = -1$, ce qui fournit un préfixe propre de poids (-1) pour le mot de Lukasiewicz u', ce qui est exclu : nous avons ainsi établi l'unicité de la décomposition

Passons à la preuve de son existence.
On a $\|w[1..1]\| = w_1 = 1$ et $\|w[1..n]\| = -1$ avec $w_n = -1$ donc $\|w[1..n-1]\| = 0$. Soit p le plus petit indice tel que $\|w[1..p]\| = 0$: on a $2 \leqslant p \leqslant n-1$. En outre, pour $1 \leqslant i < p$, on a nécessairement $\|u[1..i]\| \geqslant 1$ (car une somme négative contredirait la définition de p).
Notons $u = w[2..p]$ et $v = w[p+1..n]$. On a bien $w = (+1)\cdot u \cdot v$.
Montrons que u et v sont de Lukasiewicz.
On a bien $\|u\| = \|w[1..p]\| - 1 = 0 - 1 = -1$ et $\|v\| = \|w\| - (1 + \|u\|) = -1 - 1 + 1 = -1$.
Enfin, si $1 \leqslant i < |u| = p-1$, on a $\|u[1..i]\| = \|w[1..i+1]\| - 1 \geqslant 1 - 1 = 0$ et, si $1 \leqslant j < |v|$, on a $\|v[1..j]\| = \|w[1..j+p]\| - \|w[1..p]\| = \|w[1..j+p]\| \geqslant 0$, ce qui achève la vérification.

I.1.e On écrit le programme 2 page 2, où la fonction récursive `avance` admet pour argument `s` la somme des lettres visitées : quand elle vient à s'annuler, on peut arrêter le parcours, sinon, on continue grâce à l'appel récursif.
Même si l'énoncé ne le demande pas, on a préféré géré les cas des entrées incorrectes.

I.1.f Une solution récursive naïve revient à décomposer la longueur n sous la forme $1+k+\ell$, où k et ℓ sont impairs, de toutes les façons possibles (il y a $\left\lfloor \frac{n-1}{2} \right\rfloor$ possibilités), et pour chacune d'entre elles, de calculer récursivement l'ensemble des mots de longueur k et l'ensemble des mots de longueur ℓ, puis de faire leur produit cartésien et de calculer la réunion de ces produits. Tout cela impose de recalculer les mêmes mots un très grand nombre de fois, ce qui est très maladroit et très coûteux.

```
let décompose w =
  let rec avance s = function
      | x :: q ->
         if s+x = 0 then [x],q
         else let u,v = avance (s+x) q in x::u,v
      | _ -> failwith "le mot n'est pas de Lukasiewicz"
  in
    match w with
      | [-1] -> [],[]
      | 1 :: q -> avance 1 q
      | _ -> failwith "le mot n'est pas de Lukasiewicz" ;;
```

Programme 2 la fonction `décompose`

Une solution plus raisonnable revient à procéder de même, mais de ranger dans une table l'ensemble des mots de Lukasiewicz de longueur k dès qu'on a fini de les calculer : dans un prochain appel, il suffira de consulter la table pour retrouver le résultat, sans avoir à le recalculer.
C'est ce qu'on va faire dans la question qui suit.

I.1.g Nous écrivons tout d'abord une fonction `compose` qui calcule un genre de produit cartésien : elle renvoie la liste des mots $(+1)\cdot u\cdot v$ où u et v décrivent respectivement ses arguments `liste_U` et `liste_V`.
On a utilisé deux appels emboîtés à `it_list`, ce que d'aucuns n'aiment pas trop...
La fonction `tailleFixeLukasiewicz` (non demandée) renvoie la liste des mots de Lukasiewicz qui sont exactement de la longueur souhaitée : on remplit comme prévu une table, l'élément d'indice k de cette table contenant la liste des mots de longueur $2k+1$.
Il suffit, pour écrire la fonction demandée `obtenirLukasiewicz` de remplacer la dernière ligne de la fonction précédente par le calcul de la réunion des listes utiles de la table.
On obtient le programme 3 page 3.

Question I.2 *(Dénombrement)*

I.2.a Notons p le plus petit entier tel que $\|u[1..p]\|$ soit minimal.
Si $p=n$, c'est que pour $1\leqslant k\leqslant n-1$ on a $\|u[1..k]\|\geqslant 0$, donc u est de Lukasiewicz, et l'entier $i=1$ répond à la question.
Si $p<n$, posons $i=p+1$, $v=u[i..n]$ et $w=u[1..p]$, de sorte que $u=w\cdot v$, et montrons que $u'=v\cdot w$ est de Lukasiewicz. Remarquons que $\|w\|<-1$.
Bien sûr, $\|u'\|=\|u\|=-1$.
Soit maintenant $k\leqslant n-p$. Alors $\|u'[1..k]\|=\|v[1..k]\|=\|u[1..k]\|-\|w\|=\|u[1..k]\|-\|u[1..p]\|\geqslant 0$ par définition de p.
Soit enfin k tel que $n-p+1\leqslant k<n$. Posons $k'=k-(n-p)$: $1\leqslant k'<p$.
Alors $\|u'[1..k]\|=\|v\|+\|w[1..k']\|=\|u\|-\|u[k'+1..p]\|=-1-\|u[k'+1..p]\|>-1$ car $\|u[k'+1..p]\|=\|u[1..p]\|-\|u[1..k']\|<0$.
On a bien prouvé que u' est de Lukasiewicz.

Montrons maintenant l'unicité de cette écriture : il suffit de prouver que le seul conjugué d'une mot de Lukasiewicz u est le mot u lui-même.
Supposons en effet que $u=v\cdot w$ est de Lukasiewicz ainsi que $u'=w\cdot v$, avec $|v|\neq 0$ et $|w|\neq 0$.
Alors $\|w\|=\|u\|-\|v\|=-1-\|v\|\leqslant -1$ car le préfixe strict v de u est de poids positif ou nul : cela fournit un préfixe strict w de u' de poids strictement négatif, ce qui est la contradiction attendue.

I.2.b On propose la fonction du programme 4 page 3.
La fonction `explore` prend en arguments un triplet `(w0,v0,s0)` qui correspond au préfixe de poids minimal découvert jusqu'à présent, un couple `(w,s)` qui correspond au préfixe courant et à son poids, et enfin le suffixe courant `v`, de sorte que $u=w_0\cdot v_0=w\cdot v$ est toujours vérifié. À la fin du parcours, le résultat est alors le mot $v_0\cdot w_0$, comme on vient de le démontrer.

```
let compose liste_u liste_v =
  it_list (fun r u -> (it_list (fun s v -> ((1::u) @ v)::s) [] liste_v) @ r) [] liste_u
;;

let tailleFixeLukasiewicz n =
  if n mod 2 = 0 || n < 0 then []
  else let p = n/2 in let table = make_vect (p+1) [] in
    table.(0) <- [[-1]] ;
    for k = 1 to p do
      for j = 0 to k-1 do
       table.(k) <- table.(k) @ (compose table.(j) table.(k-1-j))
      done
    done ;
    table.(p) ;;

let rec obtenirLukasiewicz = function
    | 0 -> []
    | 1 | 2 -> [[-1]]
    | n when n < 0 -> []
    | n when n mod 2 = 0 -> obtenirLukasiewicz (n-1)
    | n -> let p = n/2 in let table = make_vect (p+1) [] in
    table.(0) <- [[-1]] ;
    for k = 1 to p do
      for j = 0 to k-1 do
       table.(k) <- table.(k) @ (compose table.(j) table.(k-1-j))
      done
    done ;
    let rec accumule i =
      if i > p then []
      else table.(i) @ (accumule (i+1))
    in
      accumule 0 ;;
```

Programme 3 la fonction `obtenirLukasiewicz`

```
let conjugue u =
  let rec explore (w0,v0,s0) (w,s) v = match v with
      | [] -> v0 @ w0
      | x :: q ->
         let s' = s+x and w' = w @ [x] in
           if s' < s0 then explore (w',q,s') (w',s') q
           else explore (w0,v0,s0) (w',s') q
  in
    explore ([],u,0) ([],0) u ;;
```

Programme 4 la fonction `conjugue`

I.2.c Montrons tout d'abord le résultat que l'énoncé propose d'admettre ; montrons par récurrence sur n la propriété $\mathcal{P}_n$ qui s'énonce : pour tous mots non vides u et v de longueur au plus égale à n tels que $u \cdot v = v \cdot u$, il existe un mot w non vide et deux entiers k et ℓ tels que $u = w^k$ et $v = w^\ell$.
La propriété $\mathcal{P}_1$ est évidemment vérifiée car si u et v sont deux lettres telles que $uv = vu$, c'est que $u = v$ et $w = u = v$, $k = \ell = 1$ répond à la question
Supposons $\mathcal{P}_n$ vérifiée pour $n \geqslant 1$ et soit u et v deux mots non vides de longueur au plus égale à $n+1$ tels que $u \cdot v = v \cdot u$. Si $|u| \leqslant n$ et $|v| \leqslant n$, l'hypothèse de récurrence conclut. Supposons désormais $|u| = n+1 \geqslant |v|$.
Si $|u| = |v| = n+1$, l'égalité $u \cdot v = v \cdot u$ entraîne $u = v$ et le choix $k = \ell = 1$ et $w = u = v$ convient.
On suppose donc maintenant que $|v| \leqslant n$.
Comme $u \cdot v = v \cdot u$, v est un préfixe de u qu'on peut écrire sous la forme $u = v \cdot u'$. Alors $v \cdot u' \cdot v = v \cdot v \cdot u'$ donc $u' \cdot v = v \cdot u'$. Si u' est le mot vide, c'est que $u = v$ et on peut choisir $k = \ell = 1$ et $w = u = v$. Sinon, $1 \leqslant |u'| = |u| - |v| \leqslant n$. Alors l'hypothèse de récurrence fournit w, k' et ℓ tels que $u' = w^{k'}$ et $v = w^\ell$, donc $u = w^{k'+\ell}$ et $v = w^\ell$.
On a bien enclenché la récurrence.

Soit $\mathcal{L}$ l'ensemble des mots de Lukasiewicz de longueur $2n+1$ et $\mathcal{M}$ l'ensemble des mots de longueur $2n+1$ et de somme -1. Bien sûr, $\mathcal{L} \subset \mathcal{M}$.
Le cardinal de $\mathcal{M}$ est simplement $\binom{2n+1}{n} = \binom{2n+1}{n+1}$ (il suffit de choisir les n occurrences de $+1$).
Appelons classe de conjugaison d'un mot u l'ensemble C_u des mots de la forme $w \cdot v$ où $u = v \cdot w$. Ces classes de conjugaisons forment une partition de $\mathcal{M}$, et on a vu que chaque classe admet parmi ses éléments un unique élément mot de Lukasiewicz : le cardinal de $\mathcal{L}$ est donc le nombre de ces classes.

Pour compter le nombre d'éléments d'une classe C_u (où u est un mot fixé de M), observons que u admet $2n+1$ préfixes stricts : ε, $u[1..1]$, $u[1..2]$, …, $u[1..2n]$. Notons $P(u)$ l'ensemble de ces préfixes.
Si $v \in P(u)$, on peut décomposer $u = v \cdot w$ (avec w non vide).
Montrons que pour deux préfixes distincts, v et v', où par exemple v est préfixe de v', écrivant $u = v \cdot w = v' \cdot w'$, les mots $w \cdot v$ et $w' \cdot v'$ sont nécessairement distincts, ce qui prouvera que C_u possède exactement $2n+1$ éléments.

Posons $v' = v \cdot z$, de sorte que $u = v \cdot z \cdot w'$, et que $w \cdot v = z \cdot w' \cdot v$ alors que $w' \cdot v' = w' \cdot v \cdot z$. Ainsi ces deux conjugués de u sont égaux si et seulement si $z \cdot (w' \cdot v) = (w' \cdot v) \cdot z$. Comme w' et z sont non vides, le résultat que l'énoncé nous propose d'admettre montre l'existence d'un mot ω non vide et de deux entiers $k \geqslant 1$ et $\ell \geqslant 1$ tels que $w' \cdot v = \omega^k$ et $z = \omega^\ell$.
Finalement $\|u\| = (k+\ell)\|\omega\| = -1$, ce qui est absurde car (-1) n'admet pas $k + \ell \geqslant 2$ comme diviseur.

On a montré que chaque classe C_u possède $2n+1$ éléments, il y a donc $\dfrac{\binom{2n+1}{n}}{2n+1}$ telles classes, et donc autant de mots de Lukasiewicz de longueur $2n+1$.

Question I.3 *(Régularité)*

I.3.a C'est un résultat bien connu : soit (Q, q_0, F, δ) un automate fini déterministe qui reconnaîtrait L. Pour tout entier k, $(+1)^k$ est le préfixe d'un mot du langage, donc $q_k = q_0.(+1)^k$ existe bien. Comme Q est fini, il existe deux entiers $1 \leqslant k < \ell$ tels que $q_k = q_\ell$. Or $q_0.(+1)^k.(-1)^{k+1} = q_k.(-1)^{k+1} = q_f$ est final car $(+1)^k(-1)^{k+1} \in L$. Mais $q_f = q_k.(-1)^{k+1} = q_\ell.(-1)^{k+1} = q_0.(+1)^\ell(-1)^{k+1}$ donc $(+1)^\ell(-1)^{k+1} \in L$, ce qui est la contradiction attendue.

I.3.b C'est là encore un résultat du cours : on peut par exemple utiliser l'automate produit de deux automates pour retrouver ce résultat.

I.3.c Observons que les mots de $L \subset \mathcal{L}$, donc $L = M \cap \mathcal{L}$ où $M = \{(+1)^n(-1)^p, (n,p) \in \mathbb{N}^2\}$. Or M est reconnaissable par l'automate de la figure 1, donc, si $\mathcal{L}$ était reconnaissable, L également. Comme L n'est pas reconnaissable, $\mathcal{L}$ ne l'est pas non plus.

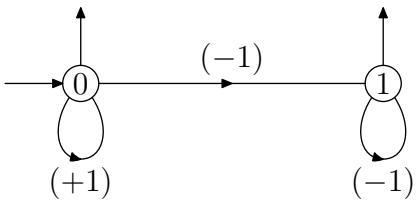

Figure 1 un automate reconnaissant M

Question I.4 *(Capsules)*

I.4.a Chaque décapsulage fait passer d'un mot de longueur n à un mot de longeur $n-2<n$. Il y a donc au maximum $|u|/2$ décapsulages consécutifs possibles sur un mot u, ce qui justifie la notation $\rho^*(u)$.

I.4.b On écrit facilement le programme 5.

```
58  let rec rho = function
59    | 1::(-1)::(-1)::q -> (-1)::q
60    | a :: (_::_::_::_ as q) -> a :: (rho q)
61    | u -> u ;;
```

Programme 5 la fonction `rho`

La ligne 59 s'intéresse à une capsule qui viendrait en tête ; la ligne 60 vérifie que derrière le `a` initial il y a au moins encore 3 lettres avant d'appeler récursivement `rho` sur la suite ; sinon, c'est qu'il n'y a pas de capsule présente.

I.4.c On écrit sans difficulté le programme 6.

```
62  let rec rhoLim u = let u' = rho u in if u <> u' then rhoLim u' else u' ;;
```

Programme 6 la fonction `rhoLim`

Pour être un peu plus efficace, on peut éviter la comparaison des deux listes `u <> u'`, en demandant à une nouvelle version `rho2` de ρ de renvoyer aussi un booléen indiquant si un décapsulage a eu lieu.
On obtient le programme 7.

```
63  let rec rho2 = function
64    | 1::(-1)::(-1)::q -> ((-1)::q,true)
65    | a :: (_::_::_::_ as q) -> let (q',b) = rho2 q in (a::q',b)
66    | u -> (u,false) ;;

67  let rec rhoLim2 u = let (u',b) = rho2 u in if b then rhoLim2 u' else u' ;;
```

Programme 7 la fonction `rhoLim` améliorée

I.4.d On remarque que

- ▷ $\|u\| = \|\rho(u)\|$ pour tout mot u ;
- ▷ (-1) est un mot de Lukasiewicz ;
- ▷ si u est de Lukasiewicz et si $u[i..i+2] = (+1)(-1)(-1)$, notant $u' = \rho(u)$, on a $u[1..i-1] = u'[1..i-1]$, donc en particulier $\|u'[1..i-1]\| = \|u[1..i-1]\| = \|u[1..i+2]\|+1 \geqslant 1$, ainsi $\|u'[1..i]\| = \|u[1..i+2]\| \geqslant 0$ et finalement pour $j > i$, $\|u'[1..j]\| = \|u[1..j+2]\|$ donc u' est également de Lukasiewicz ;
- ▷ inversement, par un raisonnement analogue, si $\rho(u) = u'$ est de Lukasiewicz, on a de même u de Lukasiewicz.

Pour conclure, il ne reste plus qu'à démontrer qu'un mot de Lukasiewicz de longueur $n \geqslant 3$ contient au moins une capsule, ce qu'on établit par récurrence.
C'est vrai pour le mot de Lukasiewicz de longueur 3 qui est la capsule $(+1)(-1)(-1)$; et tout mot de Lukasiewicz de longueur supérieure se décompose sous la forme $(+1)\cdot u\cdot v$ où u et v sont des mots de Lukasiewicz dont l'un au moins est de longueur supérieure ou égale à 3, donc contient une capsule par hypothèse de récurrence.

Partie II — Recherche de motif

Question II.1 *(Algorithme naïf)*

II.1.a On écrit naturellement le programme 8.

```
68  let coincide p m pos =
69    let np = string_length p and nm = string_length m in
70    let rec vérifie k = (k = np) || (p.[k] = m.[pos+k] && vérifie (k+1))
71    in
72      pos+np <= nm && vérifie 0 ;;
```

Programme 8 la fonction `coincide`

II.1.b On en déduit aussitôt le programme 9.

```
73  let recherche p m =
74    let np = string_length p and nm = string_length m in
75    let rec testePosition pos =
76      if pos+np > nm then []
77      else
78        if coincide p m pos then pos :: (testePosition (pos+1))
79        else testePosition (pos+1)
80    in
81      testePosition 0 ;;
```

Programme 9 la fonction `recherche`

II.1.c La fonction `coincide` peut être amenée à comparer chacune des lettres du motif dans le mot, et a donc une complexité en $|p|$ dans le pire des cas. La fonction recherche est donc de complexité $|p| \times (|m| - |p| + 1)$ dans le pire des cas, par exemple quand $p = a^k$ et $m = a\ell$.

Question II.2 *(Algorithme de Rabin-Karp)*

Remarque : la fonction `numeral` peut s'écrire de la façon suivante :

```
let numeral c = (int_of_char c) - (int_of_char '0') ;;
```

II.2.a On écrit sans difficulté la fonction `début` du programme 10, page 7. La fonction récursive auxiliaire `avance` range dans son premier argument l'entier en cours de lecture. On remarquera que la valeur numérique associée au motif p pourra être calculée par l'appel `début p np` où $np = |p|$.

La fonction `calculeDixp` calcule $10^{|p|}$. L'appel `progresse d np m c i` où $d = 10^{|p|}$ et $np = |p|$ calcule $10c + m_{i+|p|} - 10^{|p|}m_i$, suivant les recommandations de l'énoncé.

Il ne reste plus grand chose à écrire pour obtenir le programme 11, page 7.

```
82  let début m l =
83    let rec avance x i =
84      if i = l then x
85      else avance (numeral m.[i] + 10 * x) (i+1)
86    in
87      avance 0 0 ;;

88  let calculeDixp p =
89    let rec boucle x = function
90        | 0 -> x
91        | i -> boucle (10*x) (i-1)
92    in
93      boucle 1 (string_length p) ;;

94  let progresse d np m c i = 10*c + (numeral m.[i+np]) - d*(numeral m.[i]) ;;
```

Programme 10 quelques fonctions utiles

```
95  let rabinkarp p m =
96    let nm = string_length m and np = string_length p in
97    let d = calculeDixp p and px = début p np in
98    let rec testePosition c pos =
99      if pos+np = nm then if c = px then [pos] else []
100     else
101       let l = testePosition (progresse d np m c pos) (pos+1) in
102        if c = px then pos :: l else l
103   in
104     testePosition (début m np) 0 ;;
```

Programme 11 la fonction `rabinkarp`

II.2.b Pour le motif $p = 366$ et le mot $m = 97463667305$, les valeurs successives du compteur c sont, dans le choix $q = 9$: 2, 8, 4, 6, 6, 1, 7, 1 et 8, alors que $p \equiv 6\,[9]$. Il y a donc ici une seule fausse position (juste après la bonne position).
Le programme 12 est une simple modification du précédent. Les calculs s'effectuent tous modulo q, en outre le test ne s'écrit plus `c = px` mais `c = px && coincide p m pos`.

```
105  let débutMod q m l =
106    let rec avance x i =
107      if i = l then x
108      else avance ((numeral m.[i] + 10 * x) mod q) (i+1)
109    in
110      avance 0 0 ;;

111  let calculeDixpMod q p =
112    let rec boucle x = function
113        | 0 -> x
114        | i -> boucle (10*x mod q) (i-1)
115    in
116      boucle 1 (string_length p) ;;

117  let progresseMod q d np m c i =
118    (10*c + (numeral m.[i+np]) - d * (numeral m.[i])) mod q ;;

119  let rabinkarpMod q p m =
120    let nm = string_length m and np = string_length p in
121    let d = calculeDixpMod q p and px = débutMod q p np in
122    let rec testePosition c pos =
123      if pos+np = nm then if c = px && coincide p m pos then [pos] else []
124      else
125        let l = testePosition (progresseMod q d np m c pos) (pos+1) in
126         if c = px && coincide p m pos then pos :: l else l
127    in
128      testePosition (début m np) 0 ;;
```

Programme 12 la fonction `rabinkarpMod`

Dans la recherche de $p = 0001000$ dans $m = 000000000$ avec $q = 1000$, le compteur c sera constant nul, et la valeur numérique (modulo q) de p est également nulle : autrement dit, il y a fausse position à chaque instant. Il y aura donc ici $|m| - |p| + 1$ fausses positions.
Ainsi, dans le pire des cas, on devra faire $(|m| - |p| + 1)$ fois à la fonction `coincide`, ce qui donne le même coût que l'algorithme naïf.
En revanche, si on travaille sans le modulo q, l'algorithme est linéaire en la taille du mot m.
En pratique, on a donc tout intérêt à choisir un q le plus grand possible, en se limitant à la taille des entiers machine par exemple.
Si cela ne suffit pas, parce que le motif p est très long, il faudra penser à un compromis entre un q encore plus grand et la complexité introduite dans le calcul numérique en entiers longs...