Concours commun Centrale-Supelec 2008

Corrigé de l'épreuve d'informatique

Partie I - Mots de Lukasiewicz

Cette partie est la copie d'un problème publié par Jean Berstel, Jean-Eric Pin et Michel Pocchiola (Mathématiques et informatique, problèmes résolus volume 2) chez Ediscience.

I.A - Quelques propriétés

Si u est un mot sur l'alphabet $\{-1,+1\}$, la longueur de u est notée $|u|$; si k est un entier naturel au plus égal $|u|$, $s_k(u)$ est la somme des k premières lettres de u.

I.A.1) Nous avons, en notant $\mathcal{L}_n$ l'ensemble des mots de Lukasiewicz de longueur n :

$$\mathcal{L}_1 = \{(-1)\},\ \mathcal{L}_2 = \emptyset,\ \mathcal{L}_3 = \{(+1,-1,-1)\} \text{ et } \forall\, k \geq 1,\ \mathcal{L}_{2k} = \emptyset$$

puisque $\displaystyle\sum_{i=1}^{2k} u_i$ est pair pour tout mot $(u_1, u_2, \ldots, u_{2k})$ sur l'alphabet $\{-1,+1\}$.

I.A.2) La fonction `lukasiewicz : int list -> bool` s'obtient facilement en calculant itérativement les sommes partielles, jusqu'à obtenir une somme partielle négative ou une liste vide :

```
let lukasiewicz u = let sum=ref 0 and p = ref u in
while !sum >= 0 & !p<>[] do
   sum := !sum+(hd !p);
   p := tl (!p)
done;
!p=[] & !sum=(-1);;
```

On peut également utiliser une fonction récursive auxiliaire (s est la somme partielle) :

```
let lukasiewicz u =
   let rec aux s l = match l with
         [] -> s=(-1)
         |a::q -> if s=(-1) then false else aux (s+a) q in
   aux 0 u;;
```

I.A.3) Soit $w = (+1)\cdot u \cdot v$ où u et v sont deux mots de Lukasiewicz, de longueurs respectives n et m . Pour $k \in \{1, \ldots, n+m+1\}$, 5 cas sont possibles :

- si $k = 1$, $s_k(w) = 1 \geq 0$;
- si $2 \leq k \leq n$, $s_k(w) = 1 + s_{k-1}(u) \geq 1 \geq 0$;
- si $k = n+1$, $s_k(w) = 1 + s_n(u) = 0 \geq 0$;

- si $n+2 \leq k \leq n+m$, $s_k(w) = 1 + s_n(u) + s_{k-n-1}(v) = s_{k-n-1}(v) \geq 0$;
- si $k = n+m+1$, $s_k(w) = 1 + s_n(u) + s_m(v) = -1$.

On en déduit que w est un mot de Lukasiewicz.

I.A.4) Soit w un mot de Lukasiewicz de longueur $p \geq 3$. On remarque tout d'abord que $w_1 = +1$ (car $s_1(w) \geq 0$) et $s_{p-1}(w) = 0$ (car $s_p(w) = -1$ et $s_{p-1}(w) \geq 0$). On en déduit que $\{k \in [\![1, p]\!],\ s_k(w) = 0\}$ est non vide : soit p_0 son plus petit élément. En posant $u = (w_2, \ldots, w_{p_0})$ et $v = (w_{p_0+1}, \ldots, w_p)$, nous avons $w = (+1) \cdot u \cdot v$ et u et v sont des mots de Lukasiewicz :

- pour k compris entre 1 et $p_0 - 2$, $s_k(u) = s_{k+1}(w) - 1 \geq 0$ car $k+1 < p_0$;
- $s_{p_0-1}(u) = s_{p_0}(w) - 1 = -1$;
- pour k compris entre 1 et $p - p_0 - 1$, $s_k(v) = s_{p_0}(w) + s_k(v) = s_{p_0+k}(w) \geq 0$;
- $s_{p-p_0}(v) = s_p(w) = -1$.

Supposons maintenant que w admette une autre décomposition de la forme $w = (+1) \cdot u' \cdot v'$ avec u' et v' mots de Lukasiewicz. On a en particulier $s_{1+|u'|}(w) = 0$, donc $1 + |u'| \geq p_0$, soit $|u'| \geq |u|$; on en déduit que u est un préfixe de u', puis $s_{|u|}(u') = s_{|u|}(u) = -1$, ce qui impose $|u| = |u'|$, i.e. $u = u'$, puis $v = v'$: la décomposition est bien unique.

I.A.5) Pour calculer u et v, on utilise une pile p, dans laquelle on empile les lettres $w_2, w_3, \ldots, w_{p_0}$ (s contient la valeur dynamique $s_k(w)$). Cela donne la fonction `décompose : int list -> int list*int list` :

```
let decompose w = let s= ref 1 and v=ref (tl w) and p=ref [] and u=ref [] in
while !s>0 do
   s := !s+hd(!v);
   p := hd(!v)::(!p);
   v := tl(!v)
done;
while !p<> [] do
   u := hd(!p)::(!u);
   p := tl(!p)
done;
!u,!v;;
```

I.A.6) Si on écrit une fonction récursive appliquant naïvement le principe de la décomposition obtenue aux questions 3 et 4 pour construire les mots de Lukasiewicz de longueur $2n+1$, les mots de longueurs intermédiaires seront construits chacun de très nombreuses fois, puisqu'il faudra recalculer tous les mots de longueur $2k+1$, avec $k < n$, quand on aura besoin des mots de longueur $2m+1$ avec $k < m \leq n$.

La bonne méthode consiste à construire une table T de taille $n+1$, telle que l'entrée $T.(k)$ contienne la liste des mots de $\mathcal{L}_{2k+1}$: chaque mot de Lukasiewicz n'est calculé qu'un fois (voir question 7).

Cette situation est à rapprochée du calcul du n-ème terme de la suite $(x_n)_{n\geq 0}$ définie par une récurrence du type $\left(x_0 = 1;\ \forall\, n \geq 0,\ x_{n+1} = \sum_{k=0}^{n} \alpha_k x_k\right)$ où $(\alpha_k)_{k\geq 0}$ est une suite fixée. Pour $n \geq 1$, on construit la table $[x_0, x_1, \ldots, x_n]$ en temps quadratique (il faut $2k-1$ opérations pour calculer x_k), ce qui donne x_n en un temps quadratique. Par contre, le temps de calcul de la fonction naïve ci-dessous est exponentiel :

```
let rec x n = let s = ref 1 in for k=1 to (n-1) do s := !s+(alpha k)*(x k) done; !s;;
```

I.A.7) Nous utilisons l'analyse suivante :

- une table `T` de taille $n+1$ dont chaque entrée est la liste vide est créée, ainsi qu'une référence L qui pointe sur la liste contenant le seul mot $[-1]$ (une référence sur une liste sera appelée une pile) ;
- l'entrée $T.(0)$ prend la valeur $[[-1]]$: (-1) est le seul mot de Lukasiewicz de longueur 1 ;
- pour k variant de 1 à n, nous initialisons une pile vide $L1$; pour p variant de 0 à $k-1$, nous ajoutons aux piles `L` et `L1` les mots $(1)\cdot u\cdot v$, où u et v décrivent respectivement les listes `T.(p)` et `T.(k-p-1)`. À la fin de cette boucle, `L1` pointe vers la liste des mots de Lukasiewicz de longueur $2k+1$, que nous copions en `T.(k)`.
- à la fin de la boucle sur k, `L` pointe vers la liste cherchée.

Pour ajouter aux piles `L` et `L1` les nouveaux mots de Lukasiewicz, nous utiliserons deux procédures auxiliaires :

- `ajoute_mot`, appliquée à un mot m, à une liste de mots $p=[m_1;m_2;\ldots]$ et à deux piles de mots L_1 et L_2, ajoute les mots $(+1)\cdot m\cdot m_i$ aux piles L_1 et L_2 :

```
let rec ajoute_mot m p L1 L2 = match p with
   [] -> ()
   |m1::p1 -> ajoute_mot m p1 L1 L2;
              let mot=1::(m @ m1) in
                  L1 := mot::(!L1);
                  L2 := mot::(!L2);;
```

- `ajoute_liste`, appliquée à une liste de mots $p=[m_1;m_2;\ldots]$, à une liste de mots $p'=[m'_1;m'_2;\ldots]$ et à deux listes de mots L_1 et L_2, ajoute les mots $(+1)\cdot m_i\cdot m_j$ aux listes L_1 et L_2 :

```
let rec ajoute_liste p pp L1 L2 = match p with
   [] -> ()
   |m1::p1 -> ajoute_mot m1 pp L1 L2;
              ajoute_liste p1 pp L1 L2;;
```

```
let obtenirLukasiewicz n= let L = ref [[-1]] and T = make_vect (n+1) [] in
   T.(0) <- [[-1]];
   for k=1 to n do
      let L1 = ref [] in
      for p=0 to k-1 do
           ajoute_liste T.(p) T.(k-p-1) L1 L
      done;
      T.(k) <- !L1
   done;
   !L;;
```

I.B - Dénombrement

I.B.1) Remarquons tout d'abord que si u est un mot de Lukasiewicz de longueur n, aucun conjugué de u autre que u lui-même n'est un mot de Lukasiewicz. En effet, pour $i\in[\![2,n]\!]$, notons $v=(u_i,u_{i+1},\ldots,u_n,u_1,\ldots,u_{i-1})$. Nous avons

$$s_{n-i-1}(v)=\sum_{k=i}^{n}u_k=s_n(u)-s_{i-1}(u)\leq s_n(u)=-1,$$

donc v n'est pas un mot de Lukasiewicz. Ceci prouve qu'un mot u ne peut avoir qu'au plus un conjugué de Lukasiewicz.

Soit u, de longueur n, tel que $s_n(u) = -1$. Soit alors p le plus petit des entiers k pour lequel $s_k(u)$ est minimal et $v = (u_{p+1}, \ldots, u_n, u_1, \ldots, u_p)$. Si $p = n$, $s_i(u) > s_n(u) = -1$ pour tout $i < n$, donc $v = u$ est un mot de Lukasiewicz. Sinon, nous avons :

- pour $1 \leq i \leq n - p$,
$$s_i(v) = \sum_{k=p+1}^{k+i} u_k = s_{k+i}(u) - s_k(u) \geq 0$$
par minimalité de $s_k(u)$;
- pour $n - p + 1 \leq i \leq n - 1$:
$$s_i(v) = \sum_{k=p+1}^{n} u_k + \sum_{k=1}^{i+p-n} u_k = s_n(u) - s_p(u) + s_{i+p-n}(u)$$
mais $i + p - n < p$, donc $s_{i+p-n}(u) > s_p(u)$ (minimalité de p), ce qui donne $s_i(v) > -1$, i.e. $s_i(v) \geq 0$;
- $s_k(v) = s_k(u) = -1$.

Ainsi, v est dans tous les cas un mot de Lukasiewicz : tout mot dont la somme des lettres vaut -1 possède un unique conjugué de Lukasiewicz.

I.B.2) Nous aurons besoin de deux fonctions auxilaires :

- `vider : int list ref -> int -> int list ref`, appliquée à une pile p_1 qui pointe sur une liste $[m_1; \ldots; m_n]$, à un entier i compris entre 0 et n et à une pile p_2 qui pointe sur une liste $[m'_1; m'_2, \ldots; m'_p]$ dépile les i premiers éléments de p_1 et les empile dans p_2. À la fin de l'appel, p_1 et p_2 pointent respectivement sur les liste $[m_{i+1}; \cdots; m_n]$ et $[m_i; m_{i-1}; \ldots; m_1; m'_1; m'_2; \ldots : m'_p]$:
```
let rec vider p1 i p2 = match i with
   0  -> ()
   |_ ->  p2:= (hd !p1)::!p2; p1 := (tl !p1); vider p1 (i-1) p2;;
```
- `decaler : int liste -> int -> int list`, appliquée à une liste $[m_1; m_2; \ldots; m_n]$ et à un entier i compris entre 1 et $n - 1$, renvoie la liste $[m_{i+1}; m_{i+2}; \ldots; m_n; m_1; m_2; \ldots; m_i]$:
```
let decaler u i =
   let p1=ref u and p2=ref [] and p = ref [] in
      vider p1 i p2;
      vider p2 i p;
      (!p1)@(!p);;
```

La fonction `conjugue : int list -> int list` est alors facile à écrire : si u est un mot de longueur n tel que $s_n(u) = -1$, l'appel `conjugue u` :

- crée une référence `k` qui pointe sur un entier qui va parcourir $[\![1, 2, \ldots, n]\!]$;
- crée une référence `s` qui pointe sur la somme courante $s_k(u)$;
- crée une pile p qui va permettre de parcourir le mot u ;
- crée une référence `kmin` qui pointe sur la valeur de k qui minimise les sommes déjà calculée ;
- crée une référence `smin` qui pointe sur la somme minimale déjà calculée.

À la fin du calcul, `kmin` pointe sur l'entier p et `k` pointe sur la valeur n : si $p = n$, on renvoie u ; sinon, on "décale" u de p lettres, par le biais de la fonction `decaler` :

```
let conjugue u = let  k=ref 1 and s=ref (hd u) and p = ref (tl u) in
    let kmin = ref 1 and smin = ref !s in
    while !p <> [] do
      s := !s+(hd !p);
      p := tl (!p);
      k := (!k)+1;
      if !s < !smin then
        begin
        smin := !s;
        kmin := !k;
        end;
   done;
   if !kmin=(!k) then
      u
   else
      decaler u !kmin;;
```

I.B.3) Notons E l'ensemble des mots u de longueur $2n+1$ tels que $s_{2n+1}(u) = -1$. Un mot u de longueur $2n+1$ est élément de E si et seulement si il contient exactement n fois la lettre $+1$, donc E est de cardinal $\begin{pmatrix} 2n+1 \\ n \end{pmatrix}$. Pour chaque mot u de E, notons C_u l'ensemble des conjugués du mot u. Deux parties C_u et C_v, pour $u, v \in E$, sont ou bien distinctes, ou bien confondues : ces parties C_u partitionnent donc E. D'autre part, si u est un mot de E et si $i < j$ sont deux entiers compris entre 1 et $2n+1$, les mots $(u_i, \dots, u_{2n+1}, u_1, \dots u_{i-1})$ et $(u_j, \dots, u_{2n+1}, u_1, \dots u_{j-1})$ sont distincts. Dans le cas contraire, on aurait $vv' = v'v$ avec $v = (u_i, \dots, u_{j-1})$ et $v' = (u_j, \dots, u_{2n+1}, u_1, \dots u_{i-1})$. Comme v et v' sont non vides, il existerait un mot w et des entiers $k, l \geq 1$ tels que $v = w^k$ et $v' = w^l$. On en déduirait que $-1 = c_{2n+1}(u) = (k+l)\alpha$, où α est la somme des lettres de w : ce serait absurde car $\alpha \in \mathbb{Z}$ et $k + l \geq 2$.

Ainsi, les parties C_u sont de cardinal $2n+1$ et partitionnent E : comme chaque partie C_u contient un et un seul mot de Lukasiewicz, on en déduit qu'il existe exactement $\dfrac{1}{2n+1}\begin{pmatrix} 2n+1 \\ n \end{pmatrix}$ mots de Lukasiewicz de longueur $2n+1$:

$$\text{Card } \mathcal{L}_{2n+1} = \frac{(2n+1)!}{n!\,(n+1)!\,(2n+1)} = \frac{(2n)!}{n!\,(n+1)!} = \frac{1}{n+1}\begin{pmatrix} 2n \\ n \end{pmatrix}.$$

I.C - Régularité

I.C.1) Supposons que L soit reconnu par un automate déterministe complet $\mathcal{A}$. Notons Q l'ensemble des états de $\mathcal{A}$, i son état initial et F l'ensemble de ses états finals. L'application $n \longmapsto i.(+1)^n$ est alors injective de $\mathbb{N}$ dans Q : si n et m sont deux entiers naturels tels que $i.(+1)^n = i.(+1)^m$, alors :

$$i.\left((+1)^n(-1)^{m+1}\right) = \left(i.(+1)^n\right).(-1)^{m+1} = \left(i.(+1)^m\right).(-1)^{m+1} = i.\left((+1)^m(-1)^{m+1}\right) \in F$$

car $(+1)^m(-1)^{m+1}$ est élément de L : on en déduit que $(+1)^n(-1)^{m+1} \in L$, i.e. que $n = m$. Comme Q est fini, l'hypothèse faite est absurde.

I.C.2) C'est un résultat de cours.

I.C.3) Notons $\mathcal{L}$ le langage formé par les mots de Lukasiewicz et $L_1 = \{(+1)^n(-1)^m,\ n, m \in \mathbb{N}\}$. L est l'intersection de L_1 et de $\mathcal{L}$, L_1 est trivialement reconnaissable et L ne l'est pas : la contraposée du résultat précédent prouve donc que $\mathcal{L}$ n'est pas reconnaisable par automate.

On peut remarquer que la preuve donnée à la question 1) fonctionne directement avec $\mathcal{L}$.

I.D - Capsules

I.D.1) La suite $(|\rho^n(u)|)_{n\geq 0}$ est une suite décroissante d'entiers naturels : elle est donc stationnaire. Il existe donc n_0 tel que $v = \rho^{n_0}(u)$ soit de même longueur que $\rho(v)$: le mot v ne contient donc pas de capsule et $\rho(v) = v$, puis $\rho^n(u) = v$ pour tout $n \geq n_0$.

I.D.2) Si $u = (u_1, u_2, \ldots, u_n)$ est un mot, on calcule récursivement $\rho(u)$ de la façon suivante :

- si $n \leq 2$, $\rho(u) = u$;
- si $n \geq 3$ et $(u_1, u_2, u_3) = (+1, -1, -1)$, $\rho(u) = (u_3, \ldots, u_n)$;
- sinon, on pose $v = (u_2; u_3; \ldots; u_n)$ et $\rho(u) = (u_1) \cdot \rho(v)$.

```
let rec rho u = match u with
   u1::u2::u3::q -> if (u1,u2,u3)=(1,-1,-1) then
                       u3::q
                    else
                       u1::(rho (u2::u3::q))
   |_ -> u;;
```

I.D.3) Il suffit d'appliquer la fonction `rho` tant que le mot contient une capsule : le plus efficace est de modifier la fonction `rho` pour qu'elle renvoie le couple $(true, \rho(u))$ si u contient une capsule et $(false, \rho(u))$ sinon :

```
let rec rho_bis u = match u with
   u1::u2::u3::q -> if (u1,u2,u3)=(1,-1,-1) then
                       true,u3::q
                    else
                       let b,v= rho_bis (u2::u3::q) in b,u1::v
   |_ -> false,u;;


let rec rhoLim u = let b,v = rho_bis u in
   if b then
      rhoLim v
   else
      v;;
```

I.D.4) Le résultat demandé est la conséquence directes des propriétés suivantes :

- si u est un mot de Lukasiewicz, $\rho(u))$ est également un mot de Lukasiewicz. Ainsi, si u est un mot de Lukasiewicz, $\rho^*(u)$ est un mot de Lukasiewicz sans capsule ;
- si $\rho(u)$ est un mot de Lukasiewicz, u est également un mot de Lukasiewicz ;
- un mot de Lukasiewicz de longueur au moins 3 contient au moins une capsule : c'est vrai si le mot est de longueur 3, puisque $(+1, +1, -1)$ contient une capsule ; soit $n \geq 3$ et supposons que tout mot de Lukasiewicz de longueur comprise entre 3 et n possède au moins une capsule. Si u est un mot de Lukasiewicz de longueur $n+1$, on peut écrire $u = (+1) \cdot v \cdot w$ où v et w sont des mots de Lukasiewicz. Comme u est de longueur au moins 4, l'un des mots u et v est de longueur comprise entre 3 et n : il contient donc une capsule, et u également.

Partie II - Recherche de motif

II.A - Algorithme naïf

II.A.1) On obtient facilement :

```
let coincide p m pos =
   if pos+(string_length p)>string_length m then
      false
   else
      let b=ref true and i=ref 0 in
         while !i<string_length p & !b do
            b := p.[!i]=m.[!i+pos];
            i := !i+1
         done;
         !b;;
```

II.A.2) On crée une pile vide `L` puis, pour chaque position `pos` possible, on utilise la fonction `coincide` pour décider d'ajouter ou de ne pas ajouter l'entier `pos` à la pile `L` :

```
let recherche p m = let L = ref [] in
   for pos=0 to (string_length m) - (string_length p) do
      if coincide p m pos then
         L := pos::(!L)
   done;
   !L;;
```

II.A.3) Dans le pire des cas, le calcul de `coincide p m pos` demande la comparaison de toutes les lettres de p : le nombre de comparaisons est alors égal à $(|m| - |p| + 1)\,|p|$ pour chaque valeur de `pos` (on suppose que $|p| \leq |m|$). Ce cas se produit par exemple avec $p = aa\ldots a$ et $m = aa\ldots a$.

II.B - Algorithme de Rabin-Karp

II.B.1.a) On peut par exemple écrire une fonction récursive :

```
let rec init m l = match l with
   0  -> 0
   |_ -> 10*(init m (l-1))+(numeral m.[l-1]);;
```

II.B.1.b) Comme dans le A, on crée une pile vide `L`. On convertit ensuite p en un nombre entier P, puis on initialise le compteur c. Pour i variant de 0 à $|m| - |p|$, on compare c à P : en cas d'égalité, on ajoute la valeur i à la pile L. On modifie ensuite c. Cela donne :

```
let recherche_bis p m =
   let L = ref [] and l = string_length p in
   let e = puissance l and c= ref (init m l) and P = init p l in
      if !c=P then  L := 0::(!L);
      for i=0 to (string_length m)-l-1 do
```

```
      c := 10*(!c)+(numeral m.[i+l])-e*(numeral m.[i])
      if !c=P then L := (i+1)::(!L);
   done;
   !L;;
```

où `puissance l` renvoie 10^l :

```
let rec puissance l = match l with
   0 -> 1
   | _ -> 10*puissance (l-1);;
```

II.B.2.a) Comme $10 \equiv 1 \bmod 9$, le changement de compteur est $c' \leftarrow (c' + m_{i+3} - m_i)$ [9], ce qui donne le tableau :

i		0	1	2	3	4	5	6	7
m_{i+3}		6	3	6	6	7	3	0	5
m_i		9	7	4	6	3	6	6	7
c'	2	8	4	6	6	1	7	1	8

II.B.2.b) Il suffit de reprendre la procédure `recherche_bis`, mais en effectuant tous les calculs modulo q pour ne pas dépasser `max_int`. On remarquera que l'instruction `x mod q` renvoie une valeur négative quand `x` est négatif, ce qui nécessite de faire un test quand on modifie `c`.

```
let rec init_mod m l q = match l with
   0  -> 0
   |_ -> (10*(init_mod m (l-1) q)+(numeral m.[l-1])) mod q;;


let rec puissance_mod l q = match l with
   0 -> 1
   | _ -> (10*puissance_mod (l-1) q) mod q;;


let recherche_ter p m q =
   let L = ref [] and l = string_length p in
   let e = puissance_mod l q and c= ref (init_mod m l q) and P = (init_mod p l q) in
      if !c=P then  if coincide p m 0 then L := 0::(!L);
      for i=0 to (string_length m)-l-1 do
      let a=(10*(!c)+(numeral m.[i+l])-e*(numeral m.[i])) mod q in
               c := if a<0 then q+sssa else a;
      if !c=P  then if coincide p m (i+1) then L := (i+1)::(!L);
   done;
   !L;;
```

II.B.2.c) $p \equiv 0 \bmod 1000$ et $p \neq c_i = m_i \dots m_{i+6} = 0$ alors que $p \equiv c_i \bmod 1000$ pour tout i : toutes les positions sont donc des fausses positions. Lors de la recherche de p dans m, on rencontrera donc 3 fausses positions.

II.B.2.d) La question ne présente pas beaucoup d'intérêt : dans le pire des cas, on a $c_i = p$ pour tout i compris entre 0 et $|m| - |p|$ et il faut appliquer $|m| - |p| + 1$ fois la fonction `coincide`, chaque appel demandant $|p|$ comparaisons. Le temps est alors la somme des temps (à des constantes multiplicatives près) :

- $|p|$ (initialisation de `e`, `c` et `P` ;
- $|m| - |p|$ (calcul des différentes valeurs de c') ;
- $(|m| - |p| + 1)p$ (on applique $|m| - |p| + 1$ fois `coincide`).

II.B.2.e) La réponse a été donnée ci-dessus : dans le pire des cas (qui est aussi le pire des cas de l'algorithme naïf), l'algorithme de Rabin-Karp demande, en plus du calcul des différents c'_i, les mêmes calculs que l'algorithme naïf : il est donc moins efficace dans le pire des cas.

II.B.2.f) L'idée sous-jacente à l'algorithme de Rabin-Karp est que le calcul d'un nouveau c' et sa comparaison avec p (modulo q) ne prend qu'un temps constant (tant que q est assez petit pour que les calculs ne mettent en jeu que des entiers de type `int`). Le temps de calcul de l'algorithme est donc de la forme $\Theta(|m| - |p|) + \Theta(n|p|)$ où n est le nombre d'appel à la fonction `coincide`. Ainsi, il faut que n soit le plus petit possible, sachant que n est la somme du nombre de fausses positions et du nombres de "vraies" positions : pour minimiser le temps de calcul, on ne peut jouer que sur le nombre de fausses positions. Si $i \in [\![0, |m| - |p|]\!]$ est tel que $p \neq m_i m_{i+1} \dots m_{i+|p|-1}$, et en estimant que les valeurs prises par le compteur c' sont uniformément répartis dans $[\![0, \dots, q-1]\!]$, il y aura une chance sur q pour que i soit une fausse position. Ainsi, on choisira un entier q le plus grand possible pour minimiser n. Si l'on veut, comme dans l'énoncé, faire les calcul avec le type `int`, on peut choisir $q = 2^{31}$, puisque les calculs entiers se font modulo 2^{31}.

Pour utiliser un entier q plus grand, il faut définir un type d'entier modulo q avec lequel les calculs arithmétiques seront plus coûteux. Le choix d'un q optimal est donc délicat, car il nécessite de faire un compromis (quand q augmente, on diminue le nombre de fausses positions mais on augmente le temps de calcul des c').

Le meilleur temps de calcul que l'on puisse espérer obtenir est $\Theta(|m| - |p|) + \Theta(|p| \times N)$ où N est le nombre d'occurences de p dans le mot m.