Centrale 2008. Option Informatique.

Corrigé pour serveur UPS par JL. Lamard (jean-louis.lamard@prepas.org)

Partie I. Mots de Lukasiewicz.

I.A. Quelques propriétés.

Dans la suite on notera $\mathcal{L}$ l'ensemble des mots de Lukasiewicz, (P_1) la propriété $\sum_{i=1}^{n} u_1 = -1$ et (P_2) la propriété que toutes les sommes partielles strictes soient positives ou nulles.

1) On peut commencer par remarquer si $u \in \mathcal{L}$ est de longueur n et que p désigne le nombre de lettres égales à $+1$ alors $p-(n-p) = -1$ donc $n = 2p+1$. Donc tous les mots de Lukasiewicz sont de longueur impaire. □

On peut aussi remarquer que si $u \in \mathcal{L}$ et $n > 1$ (donc n impair et $n \geqslant 3$) alors d'après (P_2) (avec $k=1$) on a $u_1 = 1$ et d'après (P_2) avec $k = n-1$ et (P_1) on a $\sum_{i=1}^{n-1} u_i = 0$ et $u_n = -1$. □

Il en découle que le seul mot de Lukasiewicz de longueur 1 est (-1), que le seul de longueur 3 est $(1,-1,-1)$ et qu'il n'y pas de mots de longueur paire. □

2)

Version itérative :

```
let luka u =
  let somme_partielle = ref 0 and reste = ref u in
    while (!somme_partielle >= 0) & (!reste <> []) do
      somme_partielle := !somme_partielle + hd !reste;
      reste := tl !reste
    done;
    if (!reste <> []) then false else (!somme_partielle = -1)
;;
luka : int list -> bool = <fun>
```

Version récursive :

```
let rec luka_aux somme_partielle reste = match reste with
  | [a] -> (somme_partielle + a = -1)
  | a :: suite -> if (somme_partielle + a < 0) then false
                  else luka_aux (somme_partielle + a) suite
;;
luka_aux : int -> int list -> bool = <fun>

let luka u = match u with
  | [] -> false
  | _  -> luka_aux 0 u
;;
luka : int list -> bool = <fun>
```

Dans le pire des cas, celui d'un mot de Lukasiewicz, la version itérative et la récursive parcourent une fois toute la liste avec un temps constant à chaque élément rencontré (comparaison et addition). Donc $T(n) = O(n)$. □

3) Soient u et v deux mots de Lukasiewicz de longueurs respectives $2p+1$ et $2q+1$ et soit $w = (+1)\cdot u \cdot v$ qui est donc de longueur $n = 2p+2q+3$ (bien impaire).

Pour $1 \leqslant k \leqslant 2p+1$ on a $\sum_{i=1}^{k} w_i = 1 + \sum_{j=1}^{k-1} u_j = 1 + a > 0$ car $a \geqslant 0$ puique $u \in \mathcal{L}$ et $k-1 \leqslant 2p$.

Pour $k = 2p+2$ on a $\sum_{i=1}^{k} w_i = 1 + \sum_{j=1}^{2p+1} u_j = 1 + (-1) = 0$.

Pour $2p+3 \leqslant k \leqslant n-1$ on a $\sum_{i=1}^{k} w_i = \sum_{i=1}^{2p+2} w_i + \sum_{j=1}^{k-(2p+2)} v_l = 0 + b = b \geqslant 0$ car $v \in \mathcal{L}$ et $k-(2p+2) \leqslant 2q$.

Enfin $\sum_{i=1}^{n} w_i = 1 + (-1) + (-1) = -1$.

Ainsi si u et v appartiennent à $\mathcal{L}$ alors $(+1)\cdot u \cdot v$ appartient également à $\mathcal{L}$. □

Remarque : Il en découle que si n est impair il existe au moins un mot de Lukasiewicz de longueur n. En effet c'est vrai pour $n = 1$ et si $u \in \mathcal{L}$ est de longueur n alors $(+1)\cdot u \cdot (-1)$ appartient à $\mathcal{L}$ et est de longueur $n+2$. □

4) Unicité d'une éventuelle décomposition.
Supposons $w = (+1) \cdot u_1 \cdot v_1 = (+1) \cdot u_2 \cdot v_2$ avec $|u_1| < |u_2|$.
Soit $k = |u_1|+1$. On a $\sum_{i=1}^{k} w_i = 0$ en considérant la première décomposition et $\sum_{i=1}^{k} w_i \geqslant 1$ en considérant la seconde. Contradiction. Donc $|u_1| \geqslant |u_2|$ ce qui prouve que u_2 est un préfixe de u_1. Par raison de symétrie des rôles $u_1 = u_2$ d'où $v_1 = v_2$. □

Existence d'une telle décomposition.
Soit $w \in \mathcal{L}$ tel que $|w| = n \geqslant 3$ et K le plus petit entier compris entre 2 et $n-1$ tel que $\sum_{i=1}^{K} w_i = 0$. Un tel entier existe bien car comme déjà noté dans les remarques liminaires on a $\sum_{i=1}^{n-1} w_i = 0$ et $n-1 \geqslant 2$ car $n \geqslant 3$.
Soient $u = (w_2, \ldots, w_K) = (u_1, \ldots, u_{K-1})$ et $v = (w_{K+1}, \ldots, w_n) = (v_1, \ldots, v_{n-K})$.

- Pour $1 \leqslant k \leqslant K-2$ on a $\sum_{i=1}^{k} u_i \geqslant 0$ car $1 + \sum_{i=1}^{k} u_i = \sum_{j=1}^{k+1} w_j > 0$ par définition de K.
  Et $\sum_{i=1}^{K-1} u_i = -1$ car $1 + \sum_{i=1}^{K-1} u_i = \sum_{j=1}^{K} w_j = 0$. Donc $u \in \mathcal{L}$.
- Pour $1 \leqslant k \leqslant n-K-1$ on a $\sum_{i=1}^{k} v_i \geqslant 0$ car $1 + (-1) + \sum_{i=1}^{k} v_i = \sum_{j=1}^{K+k} w_j \geqslant 0$ car $K + k \leqslant n-1$.
  Et $\sum_{i=1}^{n-K} v_i = -1$ car $1 + (-1) + \sum_{i=1}^{n-K} v_i = \sum_{j=1}^{n} w_j = -1$. Donc $w \in \mathcal{L}$.

Ainsi se trouve établie l'existence. □

En conclusion tout mot de Lukasiewicz de longueur au moins 3 admet une unique décomposition $(+1) \cdot u \cdot v$ où u et v sont deux mots de Lukasiewicz. □

5)

On utilise la démonstration de l'existence ci-dessus pour écrire cette fonction qui maintient à jour deux listes, l'une `rev_u` étant le miroir de ce qui sera le `u` à la sortie et `v` qui sera le `v` en sortie.

```
let decompose w = match w with
  | _ when (not luka w) -> failwith "le mot n'est pas de Lukasiewicz"
  | [a]                 -> failwith "le mot est de longueur 1"
  | _ -> let S = ref 1 and rev_u = ref [] and v = ref (tl w) in
     while (!S > 0) do
       S := !S + hd !v;
       rev_u := hd !v :: !rev_u;
       v := tl !v
     done;
     (rev !rev_u,!v)
;;
decompose : int list -> int list * int list = <fun>
```

La complexité maximale est obtenue lorsque $K = n-1$ i.e. $v = (-1)$. On parcourt alors toute la liste. Les opérations sont à temps constant à chaque élément rencontré. Ainsi la complexité temporelle est-elle $O(n)$. □

6) Soit un entier $N = 2n+1$ impair avec $n \geqslant 1$. Soit p un entier variant entre 0 et $n-1$. Tout mot de Lukasiewicz de longueur N est obtenu en concaténant $(+1)$, un mot de longueur $2p+1$ et un mot de longueur $N-(2p+2) = 2q-1$ avec $p+q = n$.

Une solution récursive brutale consisterait à faire une boucle pour p variant de 0 à $n-1$ et pour chaque valeur de p à concaténer $(+1)$ avec tous les mots de longueur $2p+1$ (obtenus récursivement pour $p \geqslant 1$, cas de base $p = 0$) avec tous ceux de longueur $2q-1$ (obtenus récursivement tant que $q \geqslant 2$, cas de base $q = 1$).
Mais, situation classique avec une récursion double, les mots de longueur intermédiaire seront recalculés récursivement un grand nombre de fois.

La bonne solution consiste donc à construire un tableau (vecteur de listes d'entiers) T de longueur $n+1$ tel que la case $T.(p)$ contienne la liste de tous les mots de Lukasiewicz de longueur $2p+1$. Ainsi les mots de longueur intermédiaire ne seront calculés (avec la concaténation expliquée ci-dessus) qu'une seule fois. □

7) On commence par écrire une fonction `concatenation` qui appliquée à deux listes non vides de mots `l1` et `l2` fabrique la liste de tous les mots de la forme $(+1) \cdot u \cdot v$ avec $u \in$ `l1` et $v \in$ `l2`.

```
let concatenation l1 l2 =
  let liste = ref [] and reste1 = ref l1 in
    while (!reste1 <> []) do
      let a = hd !reste1 and reste2 = ref l2 in
        while (!reste2 <> []) do
          liste := ((1::a) @ (hd !reste2)) :: !liste;
          reste2 := tl !reste2
        done;
        reste1 := tl !reste1
    done;
    !liste
;;
concatenation : int list list -> int list list -> int list list = <fun>
```

Désormais on peut facilement écrire une fonction `tableau` qui appliquée à un entier n renvoie le tableau T défini précédemment (i.e. $T.(p)$ contient tous les mots de longueur $2p+1$) :

```
let tableau n =
  let T = make_vect (n+1) [[-1]] in
    for p=1 to n do
      T.(p) <- [];
      for k=0 to p-1 do
        T.(p) <- T.(p) @ concatenation T.(k) T.(p-k-1)
      done
    done;
    T
;;
tableau : int -> int list list vect = <fun>
```

Il reste à écrire la fonction qui renvoie la liste de tous les mots de longueur au plus m :

```
let obtenirLuka m =
  let liste = ref [[-1]]
  and n = (m-1)/2 in let T = tableau n in
    for i=1 to n do
      liste := !liste @ T.(i)
    done;
    !liste
;;
obtenirLuka : int -> int list list = <fun>
```

I.B. Dénombrement.

1) Soit un mot m de poids total -1. Alors nécessairement sa longueur n est impaire. Le cas $n = 1$ étant clair, on suppose dans la suite $n \geqslant 3$.

Unicité d'un éventuel conjugué.

Supposons que le mot m admettent deux conjugués différents u et v. Alors v est un conjugué de u donc il existe K tel que $2 \leqslant K \leqslant n$ et $(v_1, v_2, \ldots, v_n) = (u_K, u_{K+1}, \ldots, u_n, u_1, \ldots, u_{K-1})$.
On a $u_1 + \ldots + u_{K-1} \geqslant 0$ car $u \in \mathcal{L}$ et $v_1 + \ldots + v_{n+1-K} = u_K + \ldots + u_n \geqslant 0$ puisque $v \in \mathcal{L}$ et $n + 1 - K < n$.
Il en découle que u est de poids positif ou nul ce qui est en contradiction avec le fait que $u \in \mathcal{L}$. $\square$

Existence d'un conjugué.

Soit $1 \leqslant K \leqslant n$ le plus petit entier tel que $\sum_{i=1}^{K} m_i$ soit minimal.

- Si $K = n$ alors $m \in \mathcal{L}$ puisque par définition de K on a $\sum_{i=1}^{k} m_i > \sum_{i=1}^{n} m_i = -1$ pour $k < n$ donc $\sum_{i=1}^{k} m_i \geqslant 0$.
- Si $K < n$ soit $u = (m_{K+1}, \ldots, m_n, m_1, \ldots, m_K)$.
  - Pour $1 \leqslant k \leqslant n - K$ on a $\sum_{i=1}^{k} u_i = m_{K+1} + \ldots + m_{K+k} \geqslant 0$ car sinon on aurait $\sum_{i=1}^{K+k} m_i < \sum_{i=1}^{K} m_i$ ce qui est contradictoire avec la définition de K.

- Pour $n-K+1 \leqslant k \leqslant n-1$ on a $\sum_{i=1}^{k} u_i = \sum_{i=K+1}^{n} m_i + \sum_{i=1}^{k-(n-K)} m_i$. Or $k-(n-K) < K$ donc $\sum_{i=1}^{k-(n-K)} m_i > \sum_{i=1}^{K} m_i$ donc $\sum_{i=1}^{k} u_i > \sum_{i=K+1}^{n} m_i + \sum_{i=1}^{K} m_i = \sum_{i=1}^{n} m_i = -1$ donc $\sum_{i=1}^{k} u_i \geqslant 0$.
- Ainsi $u \in \mathcal{L}$.

- L'existence est ainsi établie. □

2) Compte tenu de la démonstration précédente, on commence par écrire une fonction auxiliaire qui calcule le minimum des sommes partielles d'un mot en le parcourant.
Cette fonction est de complexité temporelle $\Theta(n)$.

Puis on écrit la fonction elle-même qui parcourt le mot tant que la somme partielle n'a pas atteint la valeur minimum, en maintenant à jour deux listes : l'une nommée `rev_début` est le miroir de ce qui a déjà été parcouru et `fin` représente le reste de la liste. Lorsque la valeur minimum est atteinte, il suffit, compte tenu de la démonstration précédente, de concaténer `fin` et le miroir de `rev_début`.
Si p est la longueur de `rev_début` la complexiteé temporelle est $\Theta(p)$ pour le parcours puis $\Theta(n-p)$ pour la concaténation donc $\Theta(n)$ au total.

Ainsi la complexité temporelle du total des deux fonctions (qui sont appliquées successivement) est $\Theta(n)$.

```
let minimum m =
  let min = ref 0 and somme = ref 0 and reste = ref m in
    while (!reste <> []) do
      if (!somme + (hd !reste) < !min) then min := !somme + (hd !reste);
      somme := !somme + (hd !reste);
      reste := tl !reste
    done;
    !min
;;
minimum : int list -> int = <fun>
```

```
let conjugué m =
  let min = minimum m and somme = ref 0
  and fin = ref m and rev_début = ref [] in
    while (!somme > min) do
      somme := (hd !fin) + !somme;
      rev_début := (hd !fin) :: !rev_début;
      fin := tl !fin
    done;
    !fin @ rev(!rev_début)
;;
conjugué : int list -> int list = <fun>
```

3) Sur l'ensemble M_{2n+1} des mots de poids -1 et de longueur $2n+1$ on dira que deux mots sont conjugués si et seulement si il existe une permutation circulaire de S_{2n+1} qui permet de passer de l'un à l'autre. C'est clairement une relation d'équivalence et les classe de conjugaison forment donc une partition de M_{2n+1}.

D'après la question précédente, une classe de conjugaison donne naissance à un et un seul mot de Lukasiewicz (le conjugué d'un mot quelconque de la classe) et évidemment tout mot de Lukasiewicz peut être ainsi obtenu : c'est le conjugué de lui même.
Il en résulte que le nombre de mots de Lukasiewicz est égal au nombre de classes de conjugaison.

Or si deux mots sont conjugués ils s'écrivent respectivement $u \cdot v$ et $v \cdot u$ avec u et v non vides. Supposons qu'ils soient égaux. Alors (propriété admise) il existe un mot w non vide et deux entiers k et $\ell \geqslant 1$ tels que $u = w^k$ et $v = w^\ell$. Donc le poids de $u \cdot v$ est $(k+\ell)p$ où p est un entier égal au poids de w. Or ceci ne peut être égal à -1 car $k+\ell$ est un entier (au moins 2) et $p \in \mathbb{Z}$. Ainsi deux mots conjugués sont forcément différents.
Il en découle que toutes les classes de conjugaison ont exactement $2n+1$ éléments.

Par ailleurs le nombre de mots de M_{2n+1} est C_{2n+1}^{n} : il y a $n+1$ caractères (-1) à placer dans $2n+1$ cases.

Ainsi le nombre de classes donc des mots de Lukasiewicz de longeur $2n+1$ est $\dfrac{\mathrm{C}_{2n+1}^{n}}{2n+1} = \dfrac{\mathrm{C}_{2n}^{n}}{n+1}$ □

I.C. Régularité.

1) Supposons $L = \left\{(+1)^n(-1)^{n+1}, n \in \mathbb{N}\right\}$ reconnaissable par un automate fini $\mathcal{A} = \left(\{(+1),(-1)\}, \mathcal{Q}, p_0, F, \delta\right\}$ d'ensemble d'états $\mathcal{Q}$, d'état initial p_0, d'états finals F et de fonction de transition δ.
L'ensemble $\mathcal{Q}$ étant fini, il existe en particulier deux entiers p et q avec $p < q$ et un état x tels que, en notant $\overline{\delta}$ l'extension de δ aux mots, $\overline{\delta}(p_0,(+1)^p) = \overline{\delta}(p_0,(+1)^q)$.
Donc $\overline{\delta}(p_0,(+1)^p(-1)^{p+1}) = \overline{\delta}(p_0,(+1)^q(-1)^{p+1})$.
Supposons $\mathcal{L}$ reconnaissable. Comme $(+1)^p(-1)^{p+1} \in L$, on a ci-dessus un état final. Donc $(+1)^q(-1)^{p+1} \in L$ ce qui est impossible car $p \neq q$. □

2) Question de cours.

3) Soit $\mathcal{L}$ le langage de Lukasiewicz et soit $\mathcal{M} = \left\{(+1)^p(-1)^q, (p,q) \in \mathbb{N}^2\right\}$ qui est évidemment reconnaissable. Alors $L = \mathcal{L} \cap \mathcal{M}$ et la question précédente prouve par l'absurde que $\mathcal{L}$ n'est pas reconnaissable. □

Remarque : la démonstration de la question 1) "fonctionne " directement pour le langage de Lukasiewicz !

I.D. Capsules.

1) La suite $\left(|\rho^n(u)|\right)$ est une suite d'entiers décroissante donc évidemment constante à partir d'un certain rang. D'où le fait que la suite $\left(\rho^n(u)\right)$ est constante à partir de ce rang puisque $\rho^{k+1}(u)$ est un facteur de $\rho^k(u)$. □

En fait de manière plus précise la suite $\left(|\rho^n(u)|\right)$ commence par être strictement décroissante puis dès que $|\rho^{n_0}(u)| = |\rho^{n_0+1}(u)|$ alors $\rho^{n_0}(u)$ ne contient pas de capsules et $\rho^{n_0}(u) = \rho^*(u)$.

2) Version récursive immédiate :

```
let rec rho u = match u with
  | []                    -> u
  | [a]                   -> u
  | [a;b]                 -> u
  | a :: b :: c :: suite -> match (a,b,c) with
      | (1,-1,-1) -> c :: suite
      | _         -> a :: rho (b :: c :: suite)
;;
rho : int list -> int list = <fun>
```

3) Première version très simple :

```
let rho_lim u =
  let lim = ref u in
    while (!lim <> rho !lim) do
      lim := rho !lim
    done;
    !lim
;;
rho_lim : int list -> int list = <fun>
```

Mais cette version est coûteuse par les comparaison de listes chaînées.
Il est plus économique de modifier légèrement la fonction `rho` en `rho_bis` de manière à ce `rho_bis u` renvoie un couple `(bool,mot)` où `mot=rho u` et `bool` est vrai si et seulement si `u` ne contient pas de capsule auquel cas c'est la valeur limite.

```
let rec rho_bis u = match u with
  | []                    -> (true,u)
  | [a]                   -> (true,u)
  | [a;b]                 -> (true,u)
  | a :: b :: c :: suite -> match (a,b,c) with
      | (1,-1,-1) -> (false,c :: suite)
      | _         -> let (fini,mot) = rho_bis (b :: c :: suite) in (fini,a :: mot)
;;
rho_bis : int list -> bool * int list = <fun>
```

Il reste à appliquer cette fonction tant que `fini` est faux.

Soit itérativement :

```
let rho_lim_bis u =
  let fini = ref false and mot = ref u  in
    while (not !fini) do
      let (bool,liste) = rho_bis !mot in fini := bool; mot := liste
    done;
    !mot
;;
rho_lim_bis : int list -> int list = <fun>
```

Soit récursivement (ce qui revient exactement au même) :

```
let rec rho_lim_ter u =
  let (fini,mot) = rho_bis u in
    if fini then mot else rho_lim_ter mot
;;
rho_lim_ter : int list -> int list = <fun>
```

4) Commençons par quelques remarques :

1/ Toute somme partielle stricte de $\rho(u)$ est aussi une somme partielle stricte de u et la valeur de la somme est la même.

2/ Toute somme partielle stricte de u est supérieure ou égale à une somme partielle stricte de $\rho(u)$ (avec les notation de l'énoncé : $S_k(u) = S_k(\rho(u))$ si $k \leqslant i-1$, $S_i(u) > S_{i-1}(\rho(u))$ et $S_k(u) = S_{k-2}(\rho(u))$ pour $k \geqslant i+1$)

3/ Si $u \in \mathcal{L}$ et $|u| \geqslant 3$ alors u contient au moins une capsule. Raisonnons par récurrence sur $|u| = 2n+1$. C'est vrai pour $n = 1$ (u est alors une capsule). Supposons le résultat vrai pour tout mot de Lukasiewicz de longueur $\leqslant 2k+1$ avec $1 \leqslant k < n$. Alors la propriété de décomposition établie en I.A.4. montre que c'est vrai pour tout mot de Lukasiewicz de longueur $2n+1$.

• Soit $u \in \mathcal{L}$ avec $|u| \geqslant 3$. Alors $\rho(u) \in \mathcal{L}$ d'après 1/ ci-dessus et $|\rho(u)| < |u|$ puisque u contient une capsule d'après 3/. Il en découle immédiatement que $|\rho^*(u)| = 1$ et que $\rho^*(u) \in \mathcal{L}$. Donc $\rho^*(u) = (-1)$.

• Soit désormais u tel que $\rho^*(u) = (-1)$. Il découle de 2/ que si $\rho(v) \in \mathcal{L}$ alors $v \in \mathcal{L}$ également. Il en résulte en remontant à partir de (-1) que $u \in \mathcal{L}$.

Ainsi $u \in \mathcal{L}$ si et seulement si $\rho^*(u) = (-1)$. □

Partie II. Recherche de motifs.

II.A. Algorithme naïf.

1) Programme immédiat :

```
let coincide p m pos =
  if (pos + string_length p > string_length m) then false
  else let trouvé = ref true and k = ref 0 in
    while ((!trouvé) & (!k < string_length p)) do
      if (p.[!k] = m.[pos + !k]) then incr k
      else trouvé := false
    done;
    !trouvé
;;
coincide : string -> string -> int -> bool = <fun>
```

La complexité en nombre de comparaisons de caractères est maximale dans le cas d'occurence du motif p ($|p|$ comparaisons). Ainsi la complexité est $O(|p|)$. □

2) Là encore écriture immédiate :

```
let recherche p m =
  let liste = ref [] in
    for k = 0 to string_length m - string_length p do
      if coincide p m k then liste := k :: !liste
    done;
    !liste
;;
recherche : string -> string -> int list = <fun>
```

3) Dans le pire des cas (motif présent à chaque tour de boucle) la complexité en termes de comparaisons de caractères est de $(|m| - |p| + 1)p$

Pour $|p|$ variant entre 1 et $|m|$, cette quantité est maximale lorsque $|p| = |m|/2$.

Ainsi la complexité sera maximale pour un mot dont tous les caractères sont égaux et pour un motif de longueur moitié également constitué de cet unique caractère.
La complexité est alors équivalente à $|m|^2/2$. □

II.B. Algorithme de Rabin-Karp. (1987)

1)
a) On écrit facilement la fonction `numeral` (le code ASCII du caractère 0 étant 48) :

```
let numeral c = (int_of_char c) - 48;;
numeral : char -> int = <fun>
```

D'où clairement la fonction `initialisation` :

```
let initialisation m lg =
  if (string_length m < lg) then failwith "longueur du motif trop grande";
  let puiss = ref 1 and init = ref 0 in
    for k = lg - 1 downto 0 do
      init := !init + (!puiss * (numeral m.[k]));
      puiss := 10 * !puiss
    done;
    !init
;;
initialisation : string -> int -> int = <fun>
```

b) Puis la fonction `rabin_karp` qui suit exactement le processus décrit par l'énoncé :

```
let rabin_karp p m =
  let liste = ref []
  and c = ref (initialisation m (string_length p))
  and val_motif = initialisation p (string_length p)
  and puiss = let temp = ref 1 in
              for k = 1 to string_length motif do temp := !temp * 10 done;
              !temp
  in
    for k = 0 to string_length m - string_length p -1 do
      if (val_motif - !c = 0) then liste := k :: !liste;
      c := (10 * !c) + numeral m.[k + string_length p] - (puiss * (numeral m.[k]))
    done;
    if (val_motif - !c = 0) then liste := (string_length m - string_length p) :: !liste;
    !liste
;;
rabin_karp : string -> string -> int list = <fun>
```

2)

a) Les couples successifs (c, c') sont $(974, 2)$, $(746, 8)$, $(463, 4)$, $(636, 6)$, $(366, 6)$, $(667, 1)$, $(673, 7)$, $(730, 1)$, $(305, 8)$.

b) Il suffit de modifier la fonction précédente pour effectuer les calculs modulo q et détecter toutes positions vraies ou fausses :

```
let rabin_karp_modulo p m q =
  let liste = ref []
  and c = ref ((initialisation m (string_length p)) mod q)
  and val_motif = (initialisation p (string_length p)) mod q
  and puiss = let temp = ref 1 in
              for k = 1 to string_length motif do temp := !temp * 10 done;
              !temp
  in
    for k = 0 to string_length m - string_length p - 1 do
      if ((val_motif - !c) mod q = 0) then liste := k :: !liste;
      c := ((10 * !c) + numeral m.[k + string_length p] - (puiss * (numeral m.[k]))) mod q
    done;
    if ((val_motif - !c) mod q = 0) then
      liste := (string_length m - string_length p) :: !liste;
    !liste
;;
rabin_karp_modulo : string -> string -> int -> int list = <fun>
```

Remarque : dans les tests de comparaison on répète les modulo car $(10c' + \ell + 10^{|p|}m_i$ a toutes les chances d'être négatif et alors le représentant de la classe d'équivalence renvoyé par la fonction `mod` est négatif alors que le représentant du motif : `val_motif` est positif.

Puis il suffit de tester toutes positions possibles pour avoir le programme final :

```
let rabin_karp_bis p m q =
  let reste = ref (rabin_karp_modulo p m q) and liste = ref [] in
    while (!reste <> []) do
      if coincide p m (hd !reste) then liste := (hd !reste) :: !liste;
      reste := tl !reste
    done;
    !liste
;;
rabin_karp_bis : string -> string -> int -> int list = <fun>
```

c) Toutes les positions possibles c'est à dire 0, 1 et 2 sont bien sûr de fausses positions.

d et e)

Si l'on suppose que le calcul des restes modulo q est constant dans les $|m| - |p|$ tours de la boucle inconditionnelle de la fonction `rabin_karp_modulo`, cette fonction effectue $(|m| - |p|) \times \alpha + \beta$ opérations arithmétiques où α et β sont des constantes entières petites de l'ordre de 10 (β correspondant aux initialisations et en comptant comme des opérations arithmétiques les comparaisons modulo q) et aucune comparaison de caractères.

La fonction `rabin-karp_bis` effectue au plus $|m| - |p|$ appels à la fonction `coincide` elle même effectuant au plus $|p|$ comparaisons de caractères.

Dans le pire des cas on a donc la même complexité en comparaison de caractères qu'avec la version naïve avec en plus les opérations arithmétiques. □

e) Si l'on choisit q petit la probabilité de fausses positions sera grande d'où un nombre important d'appels à la fonction `coincide` et partant un grand nombre de comparaisons.
Si q est grand la probabilité de fausse position est faible. On aura un nombre quasi optimal d'appels à la fonction `coincide`. Si le nombre des occurences de p dans m est N, on peut espérer ainsi n'effectuer que $N|p|$ comparaisons. On peut au mieux espérer $\Theta(|m| - [p|)$ opérations arithmétiques et $\Theta(N|p|)$ comparaisons. □

Remarque : On aura bien sûr intérêt à choisir pour q une puissance de 2 afin que les calculs de reste modulo q soient rapides par simple décalage. En CAML on choisira 2^{31} si l'on utilise le type `int` puisque les calculs se font alors modulo 2^{31}.

FIN