# Partie I. Mots de Lukasiewicz

## I.1 Quelques propriétés

**Question 1.1** $(-1)$ est le seul mot de Lukasiewicz de longueur 1 ; il n'y en a pas de longueur 2, et un seul de longueur 3 : le mot $(+1,-1,-1)$.
Si $u = (u_1, u_2, \ldots, u_{2p})$ est un mot de longueur paire, la somme $\sum_{i=1}^{2p} u_i$ est paire donc $u$ ne peut être un mot de Lukasiewicz.

**Question 1.2**

```
let lukasiewicz =
  let rec aux acc = function
    | []   -> acc = -1
    | t::q -> acc >= 0 && aux (acc + t) q
  in aux 0 ;;
```

Cette fonction est de type *int list -> bool*.

**Question 1.3** Si $u$ est un mot, notons $p(u)$ la somme des lettres qui le composent (le *poids* de $u$). Un mot est de Lukasiewicz lorsque son poids est égal à $-1$ et le poids de tous ses préfixes stricts, positifs.
Considérons donc deux mots de Lukasiewicz $u$ et $v$, et posons $w = (+1)\cdot u \cdot v$.
On a $p(w) = 1 + p(u) + p(v) = 1 - 1 - 1 = -1$.
Passons maintenant en revue les différents préfixes stricts $w'$ de $w$ :

- si $w' = (+1)$ alors $p(w') = 1 \geqslant 0$ ;
- si $w' = (+1)\cdot u'$ où $u'$ est un préfixe strict de $u$, alors $p(w') = 1 + p(u') \geqslant 1$ ;
- si $w' = (+1)\cdot u$ alors $p(w') = 1 + p(u) = 0$ ;
- enfin, si $w' = (+1)\cdot u \cdot v'$ où $v'$ est un préfixe strict de $v$, alors $p(w') = 1 + p(u) + p(v') = p(v') \geqslant 0$.

Dans tous les cas on a $p(w') \geqslant 0$ donc $w$ est bien un mot de Lukasiewicz.

**Question 1.4** Soit $w$ un mot de Lukasiewicz de longueur supérieure ou égale à 3. On a $p(w_1) \geqslant 0$ donc $w_1 = (+1)$. Posons $w = (+1)\cdot w'$ et notons $u$ le plus petit préfixe strict de $w'$ vérifiant $p(u) = -1$. Un tel préfixe existe puisque $p(w'_1) \geqslant -1$ et $p(w') = -2$. Notons alors $w = (+1)\cdot u \cdot v$, et vérifions que $u$ et $v$ sont des mots de Lukasiewicz.
Par construction, $p(u) = -1$ et $p(w) = 1 + p(u) + p(v)$ donc $p(v) = p(w) = -1$.
Si $u'$ est un préfixe strict de $u$, alors $(+1)\cdot u'$ est préfixe strict de $w$ donc $p(u') \geqslant -1$. Mais par définition de $u$, $p(u')$ ne peut être égal à $-1$, donc $p(u') \geqslant 0$.
Si $v'$ est un préfixe strict de $v$, alors $(+1)\cdot u \cdot v'$ est préfixe strict de $w$ donc $1 + p(u) + p(v') \geqslant 0$ soit $p(v') \geqslant 0$.
$u$ et $v$ sont donc bien des mots de Lukasiewicz.

Supposons maintenant l'existence de deux décompositions $w = (+1)\cdot u \cdot v$ et $w = (+1)\cdot x \cdot y$. Sans perte de généralité on peut supposer que $x$ est un préfixe de $u$. Mais s'il s'agissait d'un préfixe strict de $u$ on aurait $p(x) \geqslant 0$, ce qui ne se peut. On a donc $x = u$ et par suite $y = v$. La décomposition est bien unique.

**Question 1.5** On utilise le critère obtenu à la question précédente pour caractériser $u$ :

```
let decompose w =
  let rec aux acc = function
    | s when acc = -1 -> ([], s)
    | s               -> let (u, v) = aux (acc + hd s) (tl s)
                         in (hd s)::u, v
  in aux 0 (tl w) ;;
```

Cette fonction est de type *int list -> int list * int list*.

**Question 1.6** Un algorithme récursif calculant l'ensemble des mots de longueur $2n+1$ à partir d'un appel récursif sur tous les mots de longueurs $2p+1$ et $2(n-p-1)+1$ imposerait de recalculer les mêmes mots un très grand nombre de fois et serait donc très coûteux (de coût exponentiel) ; il est préférable de procéder à une mémoïsation des mots de longueurs inférieures pour ne les calculer qu'une fois ; c'est la démarche qui est suivie dans la question suivante.

**Question 1.7** Le seul mot de Lukasiewicz de longueur 1 est égal à $(-1)$ ; tout mot de longueur $2n+1$ s'écrit de manière unique sous la forme $(+1)\cdot u\cdot v$ avec $|u|=2p+1$, $|v|=2q+1$ et $p+q=n-1$. Ainsi, pour obtenir tous les mots de longueur inférieure ou égale à $2n+1$, nous allons construire un tableau `t` de taille $n+1$, la case `t.(k)` contenant la liste des mots de taille $2k+1$.
Nous avons tout d'abord besoin d'une fonction qui à deux listes de mots $[u_1,\dots,u_p]$ et $[v_1,\dots,v_q]$ associe la liste des mots de la forme $(+1)\cdot u_i\cdot v_j$ :

```
let rec merge u v = match (u, v) with
  | [], _ -> []
  | _, [] -> merge (tl u) v
  | _, _  -> (1::(hd u)@(hd v))::(merge u (tl v)) ;;
```

Cette fonction est de type *int list list –> int list list –> int list list*.
Elle nous permet de construire le tableau `t` :

```
let tab n =
  let t = make_vect (n+1) [] in
  t.(0) <- [[-1]] ;
  for k = 1 to n do
    for p = 0 to k-1 do
      t.(k) <- t.(k) @ (merge t.(p) t.(k-1-p))
    done
  done ;
  t ;;
```

Cette fonction est de type *int –> int list list vect*.
Enfin, pour obtenir la liste des mots de Lukasiewicz il reste à réunir les cases de ce tableau :

```
let obtenirLukasiewicz n =
  let t = tab n in
  let rec aux = function
    | k when k = n+1 -> []
    | k              -> t.(k) @ (aux (k+1))
  in aux 0 ;;
```

Cette fonction est de type *int –> int list list*.

## I.2 Dénombrement

**Question 1.8** Considérons le plus petit des entiers $i\in[\![1,n]\!]$ pour lesquels $p(u_1,\dots,u_i)$ est minimal, et considérons $v=(u_{i+1},\dots,u_n,u_1,\dots,u_i)$. Nous avons déjà $p(v)=-1$ ; il reste à considérer les préfixes stricts $v'$ de $v$. Pour simplifier les notations, posons $u'=(u_1,\dots,u_i)$ et $u''=(u_{i+1},\dots,u_n)$.

- Si $v'$ est un préfixe de $u''$, alors $u'\cdot v'$ est un préfixe de $u$ et par définition de $i$, $p(u'\cdot v')\geqslant p(u')$ donc $p(v')\geqslant 0$.
- Si $v'=u''\cdot v''$, où $v''$ est un préfixe strict de $u'$, alors par définition de $i$, $p(v'')>p(u')$ donc $p(v')>p(u')+p(u'')=p(u)=-1$, et $p(v')\geqslant 0$.

De ceci il résulte que $v$ est un mot de Lukasiewicz.

Réciproquement, si $w=(u_{j+1},\dots,u_n,u_1,\dots,u_j)$ est un mot de Lukasiewicz, alors pour tout $k\in[\![j+1,n]\!]$, $p(u_{j+1},\dots,u_k)\geqslant 0$ donc $p(u_1,\dots,u_k)\geqslant p(u_1,\dots,u_j)$. Ceci prouve que $p(u_1,\dots,u_j)$ est minimal. Par définition de $i$ nous avons $i\leqslant j$ et $p(u_1,\dots,u_i)=p(u_1,\dots,u_j)$.
Mais si $i<j$ nous aurions $p(u_{i+1},\dots,u_j)=0$, et puisque $p(w)=-1$ ceci impliquerait que $p(u_{j+1},\dots,u_n,u_1,\dots,u_i)=-1$. Puisque $w$ ne peut avoir de préfixe strict de poids négatif, ceci est absurde et $i=j$, ce qui prouve l'unicité du conjugué.

**Question 1.9** Il s'agit donc de calculer le couple $(u',u'')$ de telle sorte que $p(u')$ soit minimal. L'algorithme qui suit repose sur le fait que si $u=u_1\cdot v$ avec $v=v'\cdot v''$ et $p(v')$ minimal, alors :

$$\begin{cases} u'=u_1 \text{ et } p(u')=u_1 & \text{si } p(v')\geqslant 0\\ u'=u_1\cdot v' \text{ et } p(u')=u_1+p(v') & \text{si } p(v')<0\end{cases}$$

```
let conjugue u =
  let rec aux = function
    | []   -> 0, ([], [])
    | t::q -> match aux q with
        | p, (v, w) when p < 0 -> t+p, (t::v, w)
        | _, _                 -> t, ([t], q)
  in let _, (v, w) = aux u in w @ v ;;
```

La fonction **aux** calcule le couple $\big(p(u'),(u',u'')\big)$ (avec les notations de la question précédente).
Cette fonction est de type *int list –> int list*.

**Question 1.10** Notons $\mathscr{E}$ l'ensemble des mots $u$ de longueur $2n+1$ qui vérifient $p(u)=-1$, et $\mathscr{L}$ l'ensemble des mots de Lukasiewicz de longueur $2n+1$.

$\mathscr{E}$ est l'ensemble des mots composés de $n+1$ lettres $(-1)$ et de $n$ lettres $(+1)$, donc $|\mathscr{E}| = \binom{2n+1}{n}$.

L'application qui à un mot associe son conjugué réalise une application surjective de $\mathscr{E}$ vers $\mathscr{L}$. De plus, pour tout $u \in \mathscr{L}$, l'ensemble des antécédents de $u$ est égal à l'ensemble des permutations circulaires de ses lettres. Nous allons montrer que celles-ci sont toutes distinctes, ce qui permettra d'affirmer que $u$ possède exactement $2n+1$ antécédents, et le lemme des bergers permettra de conclure que $|\mathscr{L}| = \dfrac{1}{2n+1}\binom{2n+1}{n}$.

Supposons donc qu'un mot $u \in \mathscr{E}$ possède deux permutations circulaires $v$ et $w$ identiques. Alors $w$ est aussi une permutation circulaire des lettres de $v$, donc il existe deux mots $x$ et $y$ tels que $v = x\cdot y$ et $w = y\cdot x$, et donc $x\cdot y = y\cdot x$. D'après le résultat admis, il existe un mot $z$ et deux entiers non nuls $i$ et $j$ tels que $x = z^i$ et $y = z^j$, et alors $v = z^{i+j}$. Mais dans ce cas, $p(v) = (i+j)p(z) = -1$, ce qui est absurde car $i+j \geqslant 2$ ne peut diviser $-1$.

## I.3 Capsules

**Question 1.11** La suite $\big(|\rho^n(u)|\big)_{n\in\mathbb{N}}$ est une suite d'entiers décroissante et minorée par 0, donc stationnaire. Il en est donc de même de la suite $\big(\rho^n(u)\big)_{n\in\mathbb{N}}$.

**Question 1.12**

```
let rec rho = function
  | 1::(-1)::(-1)::q -> (-1)::q
  | t::q             -> t::(rho q)
  | []               -> [] ;;
```

Cette fonction est de type *int list –> int list*.

**Question 1.13**

```
let rec rholim u =
  let v = rho u in if u = v then u else rholim v ;;
```

Cette fonction est de type *int list –> int list*.

**Question 1.14** Montrons tout d'abord que si $u$ est un mot de Lukasiewicz, il en est de même de $\rho(u)$ :

- $p(+1,-1,-1) = -1$ donc $p(\rho(u)) = p(u) = -1$.
- Notons $v$ le préfixe qui précède la première capsule de $u$ : $u = v\cdot(+1,-1,-1)\cdot w$. Alors $\rho(u) = v\cdot(-1)\cdot w$.
  Quel que soit le préfixe strict $w'$ de $w$, on a $p(v\cdot(-1)\cdot w') = p(v) - 1 + p(w') = p(v\cdot(+1,-1,-1)\cdot w') \geqslant 0$ car $u$ est un mot de Lukasiewicz. Ceci prouve que tout préfixe strict de $\rho(u)$ est de poids positif ou nul.

De ces deux points il résulte que $\rho(u)$ est encore un mot de Lukasiewicz. Par un raisonnement analogue on prouve la réciproque : si $\rho(u)$ est un mot de Lukasiewicz, il en est de même de $u$.

Montrons maintenant par récurrence sur $n \in \mathbb{N}^*$ que tout mot $u$ de Lukasiewicz de longueur $2n+1$ contient au moins une capsule :

- C'est clair lorsque $n = 1$ puisque le seul mot de Lukasiewicz vaut dans ce cas $(+1,-1,-1)$.
- Si $n \geqslant 2$ et si le résultat est acquis jusqu'au rang $n-1$, on utilise la question 1.4 : $u = (+1)\cdot v\cdot w$, où $v$ et $w$ sont deux mots de Lukasiewicz, l'un au moins étant de longueur supérieure ou égale à 3. Par hypothèse de récurrence ce dernier contient une capsule, et donc $u$ aussi.

Ainsi, si $u$ est un mot de Lukasiewicz alors $\rho^*(u)$ doit être un mot de Lukasiewicz sans capsule, autrement dit $(-1)$. Réciproquement, $(-1)$ est un mot de Lukasiewicz donc si $\rho^*(u) = -1$ alors $u$ est aussi un mot de Lukasiewicz.

# Partie II. Recherche de motif

## II.1 Algorithme naïf

**Question 2.1** On utilise le princide de l'évaluation paresseuse pour éviter des comparaisons inutiles :

```
let coincide p m pos =
  let rec aux = function
    | k when k = string_length p -> true
    | k                          -> p.[k] = m.[pos+k] && aux (k+1)
  in pos + string_length p <= string_length m && aux 0 ;;
```

Cette fonction est de type *string -> string -> int -> bool*.

**Question 2.2**

```
let recherche p m =
  let rec aux = function
    | pos when pos + string_length p > string_length m -> []
    | pos when coincide p m pos                        -> pos::(aux (pos+1))
    | pos                                              -> aux (pos+1)
  in aux 0 ;;
```

Cette fonction est de type *string -> string -> int list*.

**Question 2.3** Dans le pire des cas, le nombre total de comparaison est égal à $|p| \times (|m| - |p| + 1)$ ; c'est par exemple le cas lorsque `m = "aaaa....aaa"` et `p = "aaa...aab"`.

## II.2 Algorithme de Rabin-Karp

**Question 2.4**

```
let init m l =
  let rec aux acc = function
    | k when k = l -> acc
    | k            -> aux (10*acc + numeral m.[k]) (k+1)
  in aux 0 0 ;;
```

Cette fonction est de type *string -> int -> int*.

**Question 2.5** Pour modifier le compteur nous aurons besoin d'une fonction calculant les puissances de 10 :

```
let rec puissance = function
  | 0 -> 1
  | n -> 10 * (puissance (n-1)) ;;
```

La fonction principale s'écrit alors :

```
let rabinkarp p m =
  let l = string_length p in
  let dl = puissance l in
  let x = init p l in
  let rec aux c = function
    | i when i + l = string_length m && c = x -> [i]
    | i when i + l = string_length m          -> []
    | i when c = x -> let d = 10*c+numeral m.[i+l]-dl*numeral m.[i] in
                      i::(aux d (i+1))
    | i            -> let d = 10*c+numeral m.[i+l]-dl*numeral m.[i] in
                      aux d (i+1)
  in aux (init m l) 0 ;;
```

Cette fonction est de type *string –> string –> int list*.

**Question 2.6** Lorsque $m = 97463667305$ et $q = 9$, les différentes valeurs de $c'$ sont :

| $c$ | 974 | 746 | 463 | 636 | 366 | 667 | 673 | 730 | 305 |
|---|---|---|---|---|---|---|---|---|---|
| $c'$ | 2 | 8 | 4 | 6 | 6 | 1 | 7 | 1 | 8 |

Sachant que $p \equiv 6 \pmod 9$, il y a une seule fausse-position, pour $c = 636$.

**Question 2.7** En toute rigueur, il faut aussi réécrire les fonctions **`init`** et **`puissance`** pour tenir compte du calcul modulo $q$ et éviter tout risque de débordement :

```
let init2 q m l =
  let rec aux acc = function
    | k when k = l -> acc
    | k            -> aux ((10*acc + numeral m.[k]) mod q) (k+1)
  in aux 0 0 ;;
```

```
let rec puissance2 q = function
  | 0 -> 1
  | n -> 10 * (puissance (n-1)) mod q ;;
```

```
let rabinkarp2 q p m =
  let l = string_length p in
  let dl = puissance2 q l in
  let x = init2 q p l in
  let rec aux c = function
    | i when i + l = string_length m && c = x && coincide p m i -> [i]
    | i when i + l = string_length m                            -> []
    | i when c = x && coincide p m i ->
        let d = (10*c+numeral m.[i+l]-dl*numeral m.[i]) mod q in
          i::(aux d (i+1))
    | i                              ->
        let d = (10*c+numeral m.[i+l]-dl*numeral m.[i]) mod q in
          aux d (i+1)
  in aux (init2 q m l) 0 ;;
```

**Question 2.8** Lorsque $p = 0001000$, $m = 000000000$ et $q = 1000$, le compteur sera en permanence nul, ainsi que $p \pmod q$. Toutes les positions seront des fausses positions.

**Question 2.9** La question précédente, aisément généralisable à des mots de tailles quelconques, montre qu'il y a des cas où toutes les positions sont des fausses positions, nécessitant alors autant de comparaisons entre caractères que l'algorithme naïf. Sachant qu'il y a en plus à effectuer un certain nombre de calculs arithmétiques, l'algorithme de Rabin-Karp est dans le pire des cas moins bon que l'algorithme naïf.

**Question 2.10** L'objectif de l'algorithme de Rabin-Karp est de remplacer les comparaisons entre caractères par des calculs arithmétiques, qui ont l'avantage de se faire en coût constant. l'objectif est donc de minimiser le recours à la fonction **`coincide`** et donc de minimiser le nombre de fausses-positions.
À l'évidence de petites valeurs de $q$ augmentent le risque de fausses-positions ; on a donc tout intérêt à prendre $q$ le plus grand possible. Sachant que les entiers Caml étant calculés modulo $2^{63}$ (avec un processeur 64 bits), on choisira donc cette valeur pour $q$.

♦ ♦
♦