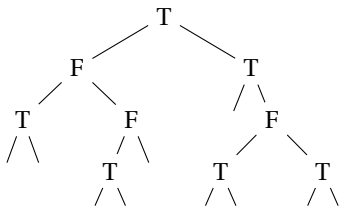
X-ENS A 2021
Facteurs dans les mots binaires

Correction proposée par Olivier Brunet.

I. Arbres de mots

Question 1 L'ensemble des mots représenté par a_2 est $\{\texttt{00},\texttt{1},\texttt{101}\}$.

L'arbre de mots réduit représentant l'ensemble $\{\varepsilon,\texttt{00},\texttt{010},\texttt{1},\texttt{110},\texttt{111}\}$ est :



Question 2 Pour montrer l'existence, on part d'un arbre binaire complet de hauteur la longueur maximale des mots à représenter en mettant `true` aux nœuds dont le chemin depuis la racine correspondant à un mot à représenter, et on le réduit en remplaçant chaque nœud de la forme $N(\texttt{false},V,V)$ par V jusqu'à ce que ce ne soit plus possible.

Pour l'unicité, considérons par l'absurde un arbre réduit a de taille minimale tel qu'il existe un autre arbre réduit b tel que $\mathbf{M}(a)=\mathbf{M}(b)$. Procédons par cas sur la forme de a :

- ⋆ si $a=V$, alors $\mathbf{M}(a)=\emptyset$ et donc $b=V$ (étant réduit lui aussi), ce qui est impossible.
- ⋆ sinon, notons $a=N(x,a_0,a_1)$. Comme a est réduit, $\mathbf{M}(a)\neq\emptyset$ et donc $b\neq V$. En notant $b=N(y,b_0,b_1)$, on a

$$x=\texttt{true}\iff\varepsilon\in\mathbf{M}(a)\iff y=\texttt{true}$$

et donc $x=y$. De plus,

$$\mathbf{M}(a_0)=\left\{m\in\{\texttt{0},\texttt{1}\}^\star\;\middle|\;\texttt{0}m\in\mathbf{M}(a)\right\}=\mathbf{M}(b_0)$$

d'où, par minimalité de a, $a_0=b_0$, et on montre de même que $a_1=b_1$.
Ainsi, on a $a=b$, ce qui est absurde.

Cela établit donc l'existence et l'unicité d'un arbre de mots réduit représentant un ensemble donné de mots.

Question 3 On peut s'aider d'une fonction auxilliaire qui, étant donné un entier n, renvoie l'arbre représentant $\{\texttt{1}^n\}$:

```
let rec uns n =
  if n = 0 then
    N (true, V, V)
  else
    N (false, V, uns (n - 1))
;;
```

On en déduit alors la fonction demandée :

```
let rec zeros_puis_uns n =
  if n = 0 then
    N (true, V, V)
  else
    N (false, zeros_puis_uns (n - 1), uns (n - 1))
;;
```

Question 4 En notant $E_{k,n}$ l'ensemble des mots de longueur au plus n et contenant exactement k caractères 1, on a :

$$\begin{aligned}\varepsilon\in E_{k,n}&\iff k=0\\ \forall\, m\in\{\texttt{0},\texttt{1}\}^\star,\quad \texttt{0}m\in E_{k,n}&\iff \left(n\geq 1\ \textbf{et}\ m\in E_{k,n-1}\right)\\ \forall\, m\in\{\texttt{0},\texttt{1}\}^\star,\quad \texttt{1}m\in E_{k,n}&\iff \left(n\geq 1\ \textbf{et}\ m\in E_{k-1,n-1}\right)\end{aligned}$$

On en déduit directement la fonction demandée, où l'on appelle `sN` lors de la construction d'arbres non vide pour s'assurer que l'on a des arbres de mots réduits :

```
let rec k_uns k n =
  if n = 0 then
    sN (k = 0, V, V)
  else
    sN (k = 0, k_uns k (n - 1), k_uns (k - 1) (n - 1))
;;
```

Question 5 Il suffit de compter le nombre de nœuds contenant `true` :

```
let rec compter = function
  | V -> 0
  | N (b, a0, a1) ->
      (if b then 1 else 0) + compter a0 + compter a1
;;
```

Question 6 Cette fonction ne présente aucune difficulté.

```
let chercher arbre mot =
  let rec aux arbre pos =
    match arbre with
    | V -> false
    | N (b, a0, a1) ->
        if pos = Array.length mot then
          b
        else if mot.(pos) = 0 then
          aux a0 (pos + 1)
        else
          aux a1 (pos + 1)
  in
  aux arbre 0
;;
```

Question 7 En s'inspirant de l'indication de l'énoncé, voici une version reposant sur une fonction auxilliaire avec un accumulateur. Cependant, contrairement à ce qui est suggéré, il est nettement préférable que la liste `pref` représente le préfixe courant dans l'ordre inverse, ce qui permet de l'allonger en temps constant en ajoutant le nouveau caractère en tête (un ajout en queue étant lui en temps linéaire) :

```
let enumerer arbre =
  let rec aux acc pref = function
    | V -> acc
    | N (b, a0, a1) ->
        let acc0 =
          if b then
            Array.of_list (List.rev pref) :: acc
          else
            acc
        in
        let acc1 = aux acc0 (0 :: pref) a0 in
        let acc2 = aux acc1 (1 :: pref) a1 in
        acc2
  in
  aux [] [] arbre
;;
```

La complexité est celle demandée, puisque si l'on omet la transformation des listes en tableaux (en temps linéaire selon la longueur de mot), chaque nœud est traité en temps constant. On note de plus que l'on a un arbre réduit, et donc chaque branche conduit à la création d'un mot.

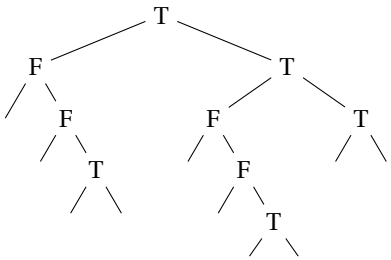
Question 8 On propose de décomposer la fonction, suivant que l'arbre est vide ou non.

```
let un_mot mot =
  let l = Array.length mot in
  let rec aux p =
    if p = l then N (true, V, V)
    else
      let a = aux (p + 1) in
      if mot.(p) = 0
      then N (false, a, V)
      else N (false, V, a)
  in
  aux 0
;;

let ajouter arbre mot =
  let l = Array.length mot in
  let rec aux arbre pos =
    match arbre with
    | N (b, a0, a1) ->
        if pos = l then
          N (true, a0, a1)
        else if mot.(pos) = 0 then
          N (b, aux a0 (pos + 1), a1)
        else
          N (b, a0, aux a1 (pos + 1))
    | V -> un_mot mot pos
  in
  aux arbre 0
;;
```

II. Arbres de suffixes

Question 9 Si $m = 1011$, l'ensemble des suffixes de m est $\{\varepsilon, 1, 11, 011, 1011\}$, d'où l'arbre de suffixes suivant :



Question 10 La hauteur de **AS**(m) est égale à longueur $\ell(m)$ du mot. De plus, on note que tous les nœuds étiquetés par `true` correspondent à des suffixes de tailles différents, et donc sont à des profondeurs différentes.

On peut donc, pour déterminer le mot d'un arbre de ses suffixes, de considérer une fonction auxilliaire qui renvoie pour chaque arbre la hauteur de la plus longue branche et le mot correspondant.

```
let retrouver_mot arbre =
  let rec aux = function
    | V -> (-1, [])
    | N (true, V, V) -> (0, [])
    | N (_, a0, a1) ->
        let t0, m0 = aux a0
        and t1, m1 = aux a1 in
        if t0 > t1 then
          (t0 + 1, 0 :: m0)
        else
          (t1 + 1, 1 :: m1)
  in
  Array.of_list (snd (aux arbre))
;;
```

Question 11 En effet, le mot étiquetant un chemin de la racine à un nœud de l'arbre des suffixes de m est le préfixe d'une branche maximale, autrement dit d'un suffixe de m (l'arbre étant réduit), c'est donc un facteur de m. De plus, deux nœuds distincts correspondront bien sûr à des facteurs distincts. Ainsi, on a une bijection entre les noeuds de **AS**(m) et les facteurs de m.

Le nombre de ces facteurs est au moins égal à $\ell(m)+1$ (puisque c'est le nombre de suffixes qui sont tous distincts, étant de longueurs différentes), et est majoré par le cardinal de $\{(i,j) \in \mathbf{N}^2 \mid 0 \leq i \leq j \leq \ell(m)\}$.

Si l'on prend le mot $m = \mathtt{0}^\ell$, alors **AS**(m) a exactement $\ell+1$ nœuds, correspondant aux facteurs de la forme $\mathtt{0}^k$ pour $0 \leq k \leq \ell$.

Si l'on considère maintenant le mot $m = \mathtt{0}^{\lfloor \ell/2 \rfloor}\mathtt{1}^{\lceil \ell/2 \rceil}$, alors pour tout $k \leq \lfloor \ell/2 \rfloor$, le mot m admet exactement $k+1$ facteurs de longueur k de la forme $\mathtt{0}^i\mathtt{1}^{k-i}$. Ainsi, m a au moins

$$\sum_{k=0}^{\lfloor \ell/2 \rfloor} (k+1) \geq \frac{\ell^2}{8}$$

facteurs.

III. Arbre des facteurs

Question 12 Pour a_5, on a (en effectuant un parcours préfixe, ce qui donne les mots dans l'ordre lexicographique, et en indiquant en parenthèses les parties correspondant aux sous-mots de la forme $m_{\lfloor p..q \rfloor}$ pour en faciliter la décomposition) :

$$\mathbf{M}(a_5) = \{\varepsilon, \mathtt{00}_{(\mathtt{1})}, \mathtt{01}_{(\mathtt{001})}, \mathtt{1}, \mathtt{10}_{(\mathtt{01})}\}$$

Ce n'est pas un arbre des facteurs de `01001` (il manque `01`), mais il est réduit car le seul nœud avec `false` a ses deux sous-arbres non vides.

Pour a_6, on a par parcours préfixe à nouveau :

$$\mathbf{M}(a_6) = \{\varepsilon, \mathtt{00}_{(\mathtt{1})}, \mathtt{01}, \mathtt{010}_{(\mathtt{01})}, \mathtt{1}, \mathtt{10}_{(\mathtt{01})}\}$$

C'est un arbre des facteurs de `01001` et il est réduit.

Enfin, pour a_7, toujours par parcours préfixe :

$$\mathbf{M}(a_7) = \{\varepsilon, \mathtt{00}_{(\mathtt{1})}, \mathtt{01}, \mathtt{010}_{(\mathtt{01})}, \mathtt{1}, \mathtt{10}_{(\mathtt{01})}\mathtt{0}_{(\mathtt{1})}\}$$

Ce n'est pas un arbre des facteurs de `01001` (à cause de `100101`) et il n'est pas réduit (à cause du nœud en `10`).

Question 13

1. Notons tout d'abord que la racine qui a ses deux sous-arbres non vides (le mot comportant au moins un `0` et au moins un `1`).

   L'arbre étant réduit, un nœud ayant au moins un sous-arbre vide contient le booléen `true`, il correspond donc à un suffixe non vide (ce n'est pas la racine) et il y a $\ell(m)$ tels suffixes.

   Notons enfin que les suffixes correspondant à des nœuds distincts sont nécessairement distincts (puisque partant de la racine, il y aura nécessairement un nœud ancêtre où l'on suivra le sous-arbre gauche dans un cas, et le sous-arbre droit dans l'autre).

   Il y a donc au plus $\ell(m)$ nœuds de a ayant au moins un sous-arbre vide.

2. Montrons plus généralement que pour tout arbre a réduit non vide, en notant $t(a)$ le nombre de ses nœuds étiquetés par `true`, on a :

$$n(a) \leq 2t(a) - 1.$$

   - ⋆ Si a est de la forme $N(b, V, V)$ alors nécessairement $b =$ `true` et donc $n(a) = t(a) = 1$.
   - ⋆ Si a est de la forme $N(b, a', V)$ ou $N(b, V, a')$ avec $a' \neq V$, alors $n(a) = 1 + n(a')$ et, comme on a vu que $b =$ `true`, $t(a) = 1 + t(a')$ d'où

$$n(a) = 1 + n(a') \leq 1 + 2t(a') - 1 = 1 + 2(t(a) - 1) - 1 \leq 2t(a) - 1$$

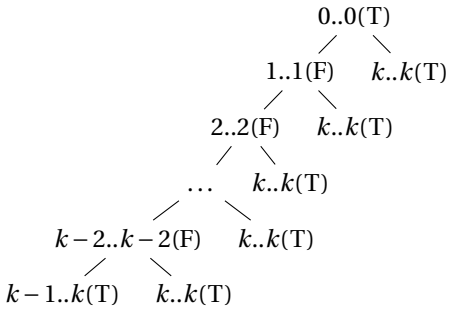
   - ⋆ Sinon, a est de la forme $N(b, a_0, a_1)$ avec $a_0 \neq V$ et $a_1 \neq V$. On a alors $n(a) = 1 + n(a_0) + n(a_1)$ et $t(a_0) + t(a_1) \leq t(a) \leq t(a_0) + t(a_1) + 1$ d'où

$$n(a) \leq 2t(a_0) + 2t(a_1) - 1 \leq 2t(a) - 1$$

   On peut améliorer l'inégalité précédente en tenant compte du fait que la racine (qui correspond au suffixe vide) contient toujours le booléen `true`. Si la racine a deux sous-arbres non vide, on a alors $t(a) = t(a_0) + t(a_1) + 1$ et donc $n(a) \leq 2t(a) - 3$. Ayant vu précédemment que $t(a) = \ell(m) + 1$ (c'est le nombre de suffixes), on a donc montré que :

$$n(a) \leq 2\ell(m) - 1$$

   Cette inégalité est valable pour tout mot contenant chacune des deux lettres. Elle est par exemple atteinte pour des mots de la forme $\mathtt{0}^k\mathtt{1}$ dont un arbre des facteurs est de la forme (en omettant les arbres vides) :



Question 14 On a vu que les nœuds de l'arbre de suffixes d'un mot m sont en bijection avec l'ensemble des facteurs de m. Ainsi, le nombre de facteurs distincts de m est égal au nombre de nœuds.

En passant aux arbres des facteurs, chaque nœud de la forme $N(p, q, b, a_0, a_1)$ correspond de même à $q - p + 1$ facteurs distincts constitués au chemin pour arriver au nœud suivi d'un préfixe de $m_{[p..q[}$. On en déduit directement le code de la fonction demandée.

```
let rec nb_facteurs = function
  | V -> 0
  | N (p, q, _, a0, a1) ->
      nb_facteurs a0 + nb_facteurs a1 + (q - p + 1)
;;
```

Question 15 On parcourt simultanément les deux mots tant que l'on ne sort pas des bornes et que les lettres correspondent.

```
let plpc mot1 p q mot2 i j =
  let k = ref 0 in
  while
    p + !k < q
    && i + !k < j
    && mot1.(p + !k) = mot2.(i + !k)
  do
    incr k
  done;
  !k
;;
```

Question 16 On parcourt f de gauche à droite et pour chaque nœud, on s'intéresse au plus long préfixe commun entre le suffixe courant de f, et le facteur $m_{[p,q[}$ du nœud courant $N(p, q, b, a_0, a_1)$.

```
let facteur arbre f =
  let l = Array.length f in
  let rec aux arbre pos =
    match arbre with
    | V -> false
    | N (p, q, _, a0, a1) ->
        let k = plpc m p q f pos l in
        if pos + k = l then (* on a lu f en entier *)
          true
        else if p + k < q then
          (* on n'a lu m[p..q[ en entier *)
          false
        else if f.(pos + k) = 0 then
          aux a0 (pos + k + 1)
        else
          aux a1 (pos + k + 1)
  in
  aux arbre 0
;;
```

La complexité est bien linéaire en $\ell(f)$ puisque chaque lettre de f est traitée en temps constant (soit dans l'appel à `plpc`, soit dans le filtre de fin).

Question 17 On s'inspire directement de la fonction `enumerer` de la question 7, en adaptant la gestion des préfixes des $m_{[p..q[}$ pour chaque nœud $N(p,q,b,a_0,a_1)$.

On obtient la fonction suivante :

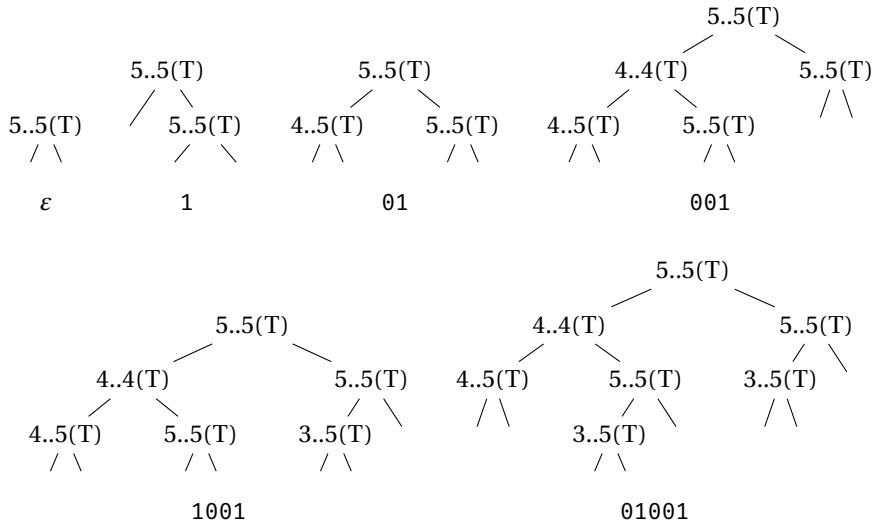
```
let facteurs arbre =
  let rec aux acc pref = function
    | V -> acc
    | N (p, q, _, a0, a1) ->
        let mpq = liste_de_sous_mot p q in
        let acc', pref' =
          List.fold_left
            (fun (acc, pref) lettre ->
              let pref' = lettre :: pref in
              (Array.of_list (List.rev pref') :: acc, pref'))
            (Array.of_list (List.rev pref) :: acc, pref)
            mpq
        in
        let acc1 = aux acc' (0 :: pref') a0 in
        let acc2 = aux acc1 (1 :: pref') a1 in
        acc2
  in
  aux [] [] arbre
;;
```

où l'on utilise une fonction `liste_de_sous_mot` permettant d'obtenir la liste des lettres de $m_{[p..q[}$:

```
let rec liste_de_sous_mot p q =
  if p = q then [] else m.(p) :: liste_de_sous_mot (p + 1) q
;;
```

De façon similaire à la fonction `enumerer`, la complexité de `facteurs` est linéaire en $\sum_{f\in\mathbf{F}(m)}\ell(f)$, puisque chaque nœud de la forme $N(p,q,b,a_0,a_1)$ est traité en $O(q-p+1)$.

Question 18 On a les arbres suivants :



Question 19 Le code, un peu fastidieux, est la traduction directe de la procédure présentée page 7.

```
let rec ajouter_suffixe i a =
  let n = Array.length m in
  match a with
  | V -> N (i, n, true, V, V)
  | N (p, q, b, a0, a1) ->
      let k = plpc m p q m i n in
      if p + k < q then
        if i + k < n then
          let aa = N (i + k + 1, n, true, V, V)
          and ab = N (p + k + 1, q, b, a0, a1) in
          if m.(i + k) = 0 then
            N (p, p + k, false, aa, ab)
          else
            N (p, p + k, false, ab, aa)
        else if m.(p + k) = 0 then
          N (p, p + k, true, N (p + k + 1, q, b, a0, a1), V)
        else
          N (p, p + k, true, V, N (p + k + 1, q, b, a0, a1))
      else if (* p + k = q *)
              i + k < n then
        if m.(i + k) = 0 then
          N (p, q, b, ajouter_suffixe (i + k + 1) a0, a1)
        else
```

```
        N (p, q, b, a0, ajouter_suffixe (i + k + 1) a1)
    else (* i + k = l(m) *)
      N (p, q, true, a0, a1)
;;
```

IV. Application : plus long facteur répété

Question 20 Une approche naïve repose sur l'utilisation de `plpc`, en déterminant pour chaque couple $0 \leq i < j < \ell(m)$ le plus long préfixe commun des suffixes $m_{[i..\ell(m)[}$ et $m_{[j..\ell(m)[}$.

```
let plfr1 mot =
  let n = Array.length mot
  and k_max = ref 0 in
  for i = 0 to n - 2 do
    for j = i + 1 to n - 1 do
      let k = plpc mot i n mot j n in
      if k > !k_max then k_max := k
    done
  done;
  !k_max
;;
```

Question 21 Pour remplir le tableau, on remarque que

$$\forall\, 0 \leq i, j < \ell(m),\ c(i+1, j+1) = \begin{cases} 1 + c(i,j) & \text{si } m_i = m_j \\ 0 & \text{sinon} \end{cases}$$

Il suffit alors de déterminer le maximum de $\{c(i,j) \mid 0 \leq i < j \leq \ell(m)\}$.

```
let plfr2 mot =
  let n = Array.length mot in
  let c = Array.make_matrix (n + 1) (n + 1) 0 in
  (* On remplit le tableau c *)
  for i = 1 to n do
    for j = 1 to n do
      if mot.(i - 1) = mot.(j - 1) then
        c.(i).(j) <- 1 + c.(i - 1).(j - 1)
    done
  done;
  (* On détermine la plus grande valeur de k *)
  let k_max = ref 0 in
  for i = 0 to n - 1 do
    for j = i + 1 to n do
      if c.(i).(j) > !k_max then k_max := c.(i).(j)
    done
  done;
  !k_max
;;
```

Question 22 D'après l'affirmation de l'énoncé, on peut écrire la fonction suivante. Les `+ 1` correspondant au caractère obtenu en allant dans le sous-arbre gauche ou droite, et les `-1` pour les arbres vides ou les nœuds non internes servent à annuler cette contribution.

```
let rec plfr3 = function
  | V -> -1
  | N (_, _, _, V, V) -> -1
  | N (p, q, _, a0, a1) ->
      q - p + 1 + max (plfr3 a0) (plfr3 a1)
;;
```

On a vu à la question 13 que la taille de l'arbre est en $O\big(\ell(m)\big)$, ce qui assure à `plfr3` une complexité linéaire en la taille du mot.

V. Mot contenant un maximum de facteurs distincts

Question 23 Un facteur de longueur k est un mot de longueur k et comme l'alphabet comporte 2 lettres, il y en a au plus 2^k. De plus, il y a $\ell(m) - k + 1$ positions de départ possible pour un facteur de longueur k, ce qui majore aussi le nombre de ces facteurs. Ainsi,

$$f_k(m) \leq \min\big(2^k, \ell(m) - k + 1\big)$$

Question 24 Si chacun des 2^k facteurs de longueur k apparaissent exactement une fois dans le mot m, alors les k premiers lettres de m fournissent un premier facteur et chaque nouvelle lettre permet d'obtenir l'un des $2^k - 1$ facteurs restant. La longueur de m est donc égale à $k + 2^k - 1$.

Question 25 Tout d'abord, si un graphe eulérien G comporte au moins une arête, alors on considère un chemin simple (c'est-à-dire n'empruntant pas deux fois une même arête) et de taille maximale. La maximalité d'un tel chemin implique que le dernier sommet visité n'a plus d'arête sortante disponible. Puisque le graphe est eulérien, il s'agit nécessairement du sommet de départ (car pour tout autre sommet, le nombre d'arêtes entrantes utilisées est égal au nombre d'arêtes sortantes utilisées plus 1), on a donc trouve un cycle C.

De plus, si l'on supprime les arêtes d'un cycle dans un graphe eulérien, on supprime pour chaque sommet un nombre égal d'arêtes entrantes et d'arêtes sortantes. Le graphe

résultant est donc encore eulérien.

Question 26 Il est clair qu'un graphe admettant un cycle eulérien est un graphe eulérien, car un tel cycle emprunte toutes les arêtes du graphe, et que pour chaque sommet, les nombres d'arêtes entrantes et d'arêtes sortantes sont égaux.

Réciproquement, si un graphe G est eulérien et fortement connexe et comporte au moins une arête, alors considérons un cycle C simple de taille maximale.

Supposons par l'absurde que C ne passe par toutes les arêtes de G. Dans ce cas, il existe un sommet s de C dont toutes les arêtes ne sont pas utilisées par C (l'existence d'un tel sommet de C découle de la forte connexité du graphe). En considérant le graphe G' obtenu à partir de G en supprimant les arêtes de C, il est clair que G' comporte au moins une arête et est eulérien. On peut donc définir un cycle C' de G' passant par s. Mais en le concaténant à C, on obtient un cycle simple de G strictement plus grand, ce qui est absurde.

En conclusion, un cycle simple de G de taille maximale est nécessairement eulérien.

Question 27 Le graphe G_k est eulérien, puisque les degrés entrant et sortant de chaque sommet sont égaux à 2. De plus, le graphe est fortement connexe, puisque étant donné deux sommets $s = s_1 \dots s_{k-1}$ et $t = t_1 \dots t_{k-1}$, il existe un chemin allant de s à t. Ce chemin passe par les sommets correspondant aux facteurs de longueur $k-1$ de $st = s_1 \dots s_{k-1} t_1 \dots t_{k-1}$.

D'après la question précédente, G_k admet donc un cycle eulérien. Or, comme les arêtes de G_k sont en bijection avec les mots de longueur k, on en déduit que le mot obtenu à partir d'un cycle eulérien en fixant un sommet qui sera le début du mot, et en ajoutant pour chaque arête la lettre final du sommet d'arrivée, on obtient un mot comportant chaque facteur de longueur k exactement une fois.