

**ECOLE POLYTECHNIQUE - ECOLES NORMALES SUPERIEURES**

**CONCOURS D'ADMISSION 2019**

**VENDREDI 19 AVRIL 2019 - 14h00 – 18h00**

**FILIERE MP (Spécialité Informatique)**

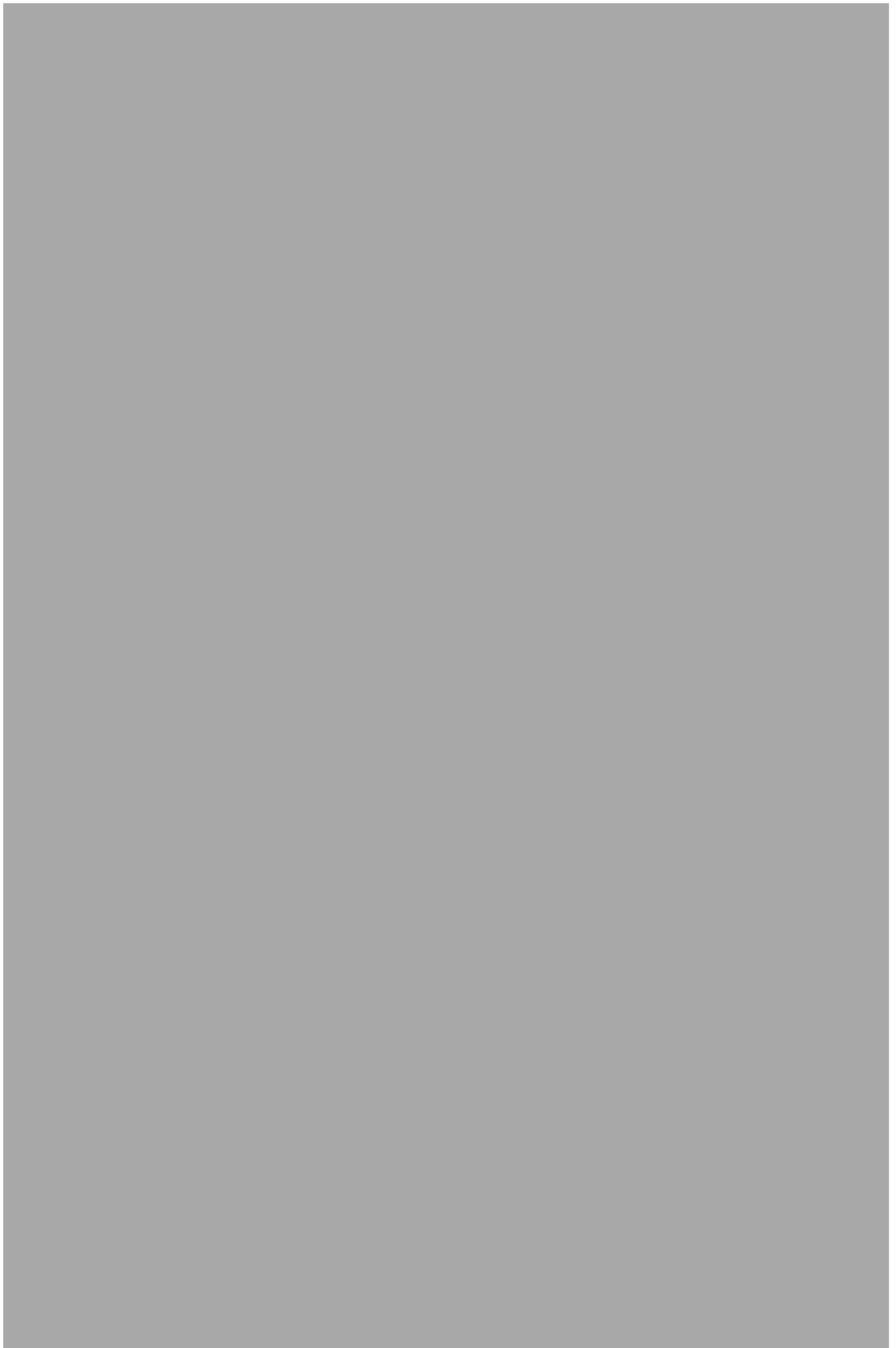
**Epreuve n° 4**

**INFORMATIQUE A**

**(XULCR)**

*Durée : 4 heures*

*L'utilisation des calculatrices n'est pas autorisée pour cette épreuve*



L'utilisation des calculatrices **n'est pas autorisée** pour cette épreuve.  
Le langage de programmation sera **obligatoirement OCaml**.

---

## Autour des sous-mots et des sur-mots

---

### Préliminaires

Un mot est une suite de lettres  $a_0 \cdots a_{n-1}$  tirées d'un alphabet fini  $A = \{\mathbf{a}, \mathbf{b}, \dots\}$ . On utilisera  $u, v, u', u'', u_1, u_2, \dots$  pour dénoter les éléments de  $A^*$ , c.-à-d. les mots sur  $A$ . On note  $\varepsilon$  pour le mot vide et  $|u|$  pour la longueur de  $u$ , de sorte que  $|\varepsilon| = 0$ .

Si un mot  $u$  se décompose sous la forme  $u = u_1 v u_2$ , alors  $v$  est un **facteur** de  $u$ , et même un préfixe (ou un suffixe) si  $u_1 = \varepsilon$  (ou si  $u_2 = \varepsilon$ ) dans cette décomposition. Dans le cas d'un mot  $u = a_0 \cdots a_{n-1}$  on écrit «  $u[i, j[$  », sous la condition  $0 \leq i \leq j \leq n$ , pour désigner le facteur  $a_i \cdots a_{j-1}$ . Cette notation s'étend à  $u[i \cdots [$  et  $u[i]$  pour désigner, respectivement,  $u[i, n[$  et  $u[i, i + 1[$ .

Ce que l'on appelle sous-mot de  $u$  correspond à la notion classique de sous-suite, ou de suite extraite, et ne doit pas être confondu avec un facteur. Pour  $u = a_0 \cdots a_{n-1}$ , on dira qu'un mot  $v$  de longueur  $m$  est un **sous-mot** de  $u$ , ce que l'on notera  $v \preceq u$ , s'il existe une suite strictement croissante  $0 \leq p_0 < p_1 < \cdots < p_{m-1} < n$  telle que  $v = a_{p_0} a_{p_1} \cdots a_{p_{m-1}}$ . Par exemple, `caml`  $\preceq$  `bechamel`. Formellement, pour tout  $n \in \mathbb{N}$ , nous noterons  $[n]$  pour l'ensemble  $\{0, 1, 2, \dots, n-1\}$ , de sorte que la suite  $p_0, p_1, \dots, p_{m-1}$  peut être vue comme une application strictement croissante  $p : [m] \rightarrow [n]$ . Pour une telle application, on note  $v = u \circ p$  pour dire que  $v$  est le sous-mot extrait de  $u$  via  $p$  et on dit que  $p$  est un **plongement** de  $v$  dans  $u$ , noté  $p : v \preceq u$ . Notons qu'il peut exister plusieurs façons différentes de plonger  $v$  dans  $u$ .

Notre objectif ici est de développer des algorithmes impliquant à divers titres la notion de sous-mot : recherche d'un sous-mot à l'intérieur d'un texte, dénombrement des sous-mots, raisonnement sur l'ensemble des sous-mots d'un texte ou d'un langage.

**Complexité.** Par **complexité en temps** d'un algorithme  $A$  on entend le nombre d'opérations élémentaires (comparaison, addition, soustraction, multiplication, division, affectation, test, etc.) nécessaires à l'exécution de  $A$  dans le cas le pire. Par **complexité en espace** d'un algorithme  $A$  on entend l'espace mémoire minimal nécessaire à l'exécution de  $A$  dans le cas le pire. Lorsque la complexité en temps ou en espace dépend d'un ou plusieurs paramètres  $\kappa_0, \dots, \kappa_{r-1}$ , on dit que  $A$  a une complexité en  $\mathcal{O}(f(\kappa_0, \dots, \kappa_{r-1}))$  s'il existe une constante  $C > 0$  telle que, pour toutes les valeurs de  $\kappa_0, \dots, \kappa_{r-1}$  suffisamment grandes (c'est-à-dire plus grandes qu'un certain seuil), pour toute instance du problème de paramètres  $\kappa_0, \dots, \kappa_{r-1}$ , la complexité est au plus  $C f(\kappa_0, \dots, \kappa_{r-1})$ .

**OCaml.** On rappelle quelques éléments du langage OCaml qui peuvent être utiles. Une chaîne de caractères `s` a le type `string`, sa longueur est obtenue avec `String.length s` et son  $i$ -ième caractère avec `s.[i]`, les caractères étant indexés à partir de 0. Un tableau `t` a le type `array`, où  $\tau$  est le type des éléments, et sa longueur est obtenue avec `Array.length t`. Son  $i$ -ième élément est obtenu avec `t.(i)` et modifié avec `t.(i) <- val`, les éléments étant indexés à partir de 0. L'expression `Array.make n val` construit un tableau de taille `n` dont les éléments sont initialisés avec la valeur `val`. En OCaml, une matrice est un tableau de tableaux de même taille. L'expression `Array.make_matrix n m val` construit une matrice de `n` lignes et `m` colonnes, dont les éléments sont tous initialisés avec la valeur `val`. Le candidat est libre d'utiliser tout autre élément du langage OCaml et de sa bibliothèque standard.

**Question 1. a.** Montrez que pour deux mots  $u$  et  $u'$  et deux lettres  $a$  et  $a'$ , on a l'équivalence suivante :

$$ua \preceq u'a' \iff ua \preceq u' \text{ ou } (a = a' \text{ et } u \preceq u'). \quad (1)$$

**b.** Programmez une fonction OCaml `teste_sous_mot : string -> string -> bool` décidant en temps polynomial si un mot  $v$  est sous-mot d'un mot  $u$ . Détaillez et justifiez votre analyse de complexité.

## I. Compter et construire

On note  $\binom{u}{v}$  le nombre de plongements de  $v$  dans  $u$ , de sorte que  $v \preceq u$  si et seulement si  $\binom{u}{v} > 0$ . Notons en particulier que  $\binom{u}{\varepsilon} = 1$  pour tout mot  $u \in A^*$  car il n'existe qu'une injection de  $[0]$ , c.-à-d.  $\emptyset$ , dans  $\{0, 1, \dots, |u| - 1\}$  et cette injection est bien un plongement.

**Question 2. a.** Montrez que  $\binom{abab}{ab} = 3$ .

**b.** Que vaut  $\binom{a^n}{a^m}$  quand  $a \in A$  est une lettre ?

On rappelle que  $a^n$  est le mot constitué de  $n$  occurrences de la lettre  $a$ .

**c.** Montrez que  $\binom{ua}{va} = \binom{u}{va} + \binom{u}{v}$  pour tous mots  $u, v \in A^*$  et toute lettre  $a \in A$ .

**Question 3.** Pour calculer  $\binom{u}{v}$  on considère la fonction OCaml suivante.

```
let nb_plongements (v:string) (u:string) =
  let rec aux i j =
    if i = 0 then 1
    else if j = 0 then 0
    else if v.[i-1] = u.[j-1] then (aux (i - 1) (j - 1)) + (aux i (j - 1))
    else aux i (j - 1)
  in
  aux (String.length v) (String.length u)
```

**a.** Prouvez sa terminaison.

**b.** Justifiez sa correction, c.-à-d., expliquez pourquoi elle renvoie bien la valeur  $\binom{u}{v}$ .

On note  $T(v, u)$  le nombre de fois où la fonction `aux` est appelée lors du calcul de `nb_plongements v u`.

- Question 4.** **a.** Montrez qu'il existe une constante  $C_1$  telle que  $T(v, u) < 2^{|u|} \cdot C_1$ .  
**b.** Montrez que l'on ne peut pas majorer  $T(v, u)$  par une fonction polynomiale de  $\binom{u}{v}$ .  
**c.** Montrez qu'il existe une constante  $C_2$  telle que  $T(v, u) \geq 2\binom{u}{v} + C_2$ .

La question précédente a montré que la fonction `nb_plongements` proposée dans le sujet demande un temps de calcul parfois exponentiel en la taille  $|u| + |v|$  de ses arguments. De meilleurs algorithmes existent...

- Question 5.** Programmez en OCaml une nouvelle fonction `nb_plongements_rapide : string -> string -> int` qui calcule  $\binom{u}{v}$  en temps polynomial en  $|u| + |v|$ . Détaillez votre analyse de complexité en temps et en espace.  
 Indication : on pourra utiliser la programmation dynamique.

On cherche maintenant à dénombrer les sous-mots d'un mot  $u$ . On note  $\downarrow u$  pour  $\{v \mid v \preceq u\}$ . Il s'agit d'un langage fini. Par exemple  $\downarrow \text{abab} = \{\varepsilon, \text{a}, \text{b}, \text{ab}, \text{aa}, \text{ba}, \text{bb}, \text{aab}, \text{aba}, \text{abb}, \text{bab}, \text{abab}\}$  de sorte que `abab` a 12 sous-mots distincts, ce que l'on note  $\text{Card}(\downarrow \text{abab}) = 12$ .

Les langages étant des ensembles (des parties  $L, L', \dots$  de  $A^*$ ), on utilisera les notations  $L \cup L'$ ,  $L \setminus L'$ , etc. avec leur signification ensembliste habituelle. On utilise aussi la notation  $L \cdot L'$  pour désigner le produit de concaténation de deux langages :  $L \cdot L' = \{uv \mid u \in L, v \in L'\}$ . Dans le cas d'un singleton  $L = \{u\}$ , on écrit souvent  $u \cdot L'$  au lieu de  $\{u\} \cdot L'$ .

- Question 6.** **a.** Montrez que, pour tous mots  $v, w$  et toute lettre  $a$ , on a

$$\downarrow wava = \downarrow wav \cup (\downarrow wav \setminus \downarrow w) \cdot a. \quad (\ddagger)$$

- b.** Montrer que l'union  $\downarrow wav \cup (\downarrow wav \setminus \downarrow w) \cdot a$  est disjointe si et seulement si le mot  $v$  ne contient pas la lettre  $a$ .

Quand l'union est disjointe dans l'équation  $(\ddagger)$ , on peut obtenir  $\text{Card}(\downarrow u)$ , pour  $u = wava$ , en combinant  $\text{Card}(\downarrow wav)$  et  $\text{Card}(\downarrow w)$ . Cette approche se généralise au cas d'un mot  $u$  quelconque.

- Question 7.** **a.** Donnez des équations récursives permettant de calculer  $\text{Card}(\downarrow u)$  en se ramenant à des préfixes de  $u$ . On pourra considérer par exemple les diverses occurrences de la dernière lettre de  $u$  quand elle existe.  
**b.** En se basant sur vos équations, programmez une fonction OCaml `nb_sousmots : string -> int` qui, pour un mot  $u$  donné, calcule  $\text{Card}(\downarrow u)$  en temps polynomial en  $|u|$ . Justifiez votre analyse de complexité.

Un **sur-mot** commun à  $u$  et  $v$  est un mot  $w$  tel que  $u \preceq w$  et  $v \preceq w$ . Il existe une infinité de tels mots. Parmi tous ces sur-mots communs à  $u$  et  $v$ , on s'intéresse à celui qui est le plus court, et qui est le premier dans l'ordre lexicographique pour départager les sur-mots communs de même longueur. Ce mot est noté  $\text{pcsmc}(u, v)$  et par exemple  $\text{pcsmc}(\text{informatique}, \text{difficile}) = \text{difnfcormatilque}$ .

**Question 8. a.** Soient  $a, b$  deux lettres distinctes. Montrez que si  $\text{pcsmc}(ua, vb) = wa$  alors  $w = \text{pcsmc}(u, vb)$ , ceci pour tous mots  $u, v, w$ .

**b.** Généralisez la propriété précédente en donnant des équations qui permettent de caractériser  $\text{pcsmc}(ua, vb)$  dans le cas général, y compris quand  $a = b$ .

**c.** Programmez une fonction OCaml calculant  $\text{pcsmc}(u, v)$  en temps polynomial en  $|u| + |v|$  pour des mots  $u$  et  $v$  arbitraires. Détaillez votre analyse de complexité.

## II. Sous-mots et expressions rationnelles

On rappelle que les **expressions rationnelles** sont écrites à partir des expressions de base «  $\varepsilon$  », «  $\emptyset$  », ainsi que les lettres «  $a$  », «  $b$  »,  $\dots$ , que l'on peut combiner au moyen des opérateurs binaires «  $+$  » et «  $\cdot$  » (désignant l'union et la concaténation de langages) ainsi que de l'« étoile de Kleene », un opérateur unaire «  $*$  » noté en exposant.

Le langage représenté par une expression rationnelle  $e$  est défini inductivement par  $L(\varepsilon) = \{\varepsilon\}$ ,  $L(\emptyset) = \emptyset$ ,  $L(a) = \{a\}$ ,  $\dots$ ,  $L(e + e') = L(e) \cup L(e')$ ,  $L(e \cdot e') = L(e) \cdot L(e')$  et enfin

$$L(e^*) = L(e)^* = \{\varepsilon\} \cup L(e) \cup L(e) \cdot L(e) \cup \dots = \bigcup_{i \in \mathbb{N}} \overbrace{L(e) \cdot L(e) \cdots L(e)}^i.$$

Pour manipuler des expressions rationnelles, on utilisera la définition OCaml suivante :

```
type ratexp =
  | Epsilon
  | Empty
  | Letter of char
  | Sum of ratexp * ratexp
  | Product of ratexp * ratexp
  | Star of ratexp
```

Par exemple, les expressions rationnelles  $a \cdot (b + c)^*$  et  $((\emptyset + \varepsilon)^*)^*$  seront représentées par

```
let e_exmp1 = Product (Letter 'a', Star (Sum (Letter 'b', Letter 'c')))
let e_exmp2 = Star (Star (Sum (Empty, Epsilon)))
```

La taille d'une expression rationnelle, notée  $|e|$ , est le nombre de constructeurs apparaissant dans l'expression. On pourrait calculer  $|e|$  en OCaml au moyen du code suivant :

```
let rec taille_ratexp (e : ratexp) =
  match e with
  | Empty -> 1 | Epsilon -> 1 | Letter _ -> 1
  | Sum (e1,e2) -> 1 + taille_ratexp(e1) + taille_ratexp(e2)
  | Product (e1,e2) -> 1 + taille_ratexp(e1) + taille_ratexp(e2)
  | Star (e1) -> 1 + taille_ratexp(e1)
```

**Question 9.** On définit les expressions rationnelles  $e_1$  et  $e_2$  par

```
let e1 = Product (Star (Product (Sum (Letter 'a', Empty), Letter 'c')),
  Product (Letter 'b',
    Product (Empty,
      Star (Product (Letter 'c', Letter 'c')))))
let e2 = Product (Star(Product(Letter 'b', Letter 'a')),
  Product (Sum (Epsilon, Letter 'a'), Star(Letter 'c')))
```

Pour chacun des langages  $L(e_1)$  et  $L(e_2)$ , dites s'il contient un mot commençant par a ; par b ; par c.

**Question 10.** Programmez une fonction OCaml `peut_debuter_par : ratexp -> char -> bool` testant, pour une expression rationnelle  $e$  et une lettre  $a$ , si  $L(e)$  contient un mot commençant par  $a$ .

Pour un langage  $L$ , on définit  $\downarrow L = \bigcup_{w \in L} \downarrow w$ . On s'intéresse maintenant à la question de savoir, pour un mot  $u$  et une expression rationnelle  $e$ , si  $u$  est dans  $\downarrow L(e)$ , c.-à-d. si  $u$  est sous-mot d'un des mots définis par  $e$ . Une solution possible passe par des calculs de résidus de langages. Formellement, pour un langage  $L \subseteq A^*$  et un mot  $u \in A^*$ , on définit le **résidu de  $L$  par  $u$**  comme

$$\langle u \rangle L = \{v \in A^* \mid \exists w \text{ tel que } u \preceq w \text{ et } vw \in L\}.$$

Ainsi,  $u$  est sous-mot d'un mot de  $L$  si et seulement si  $\varepsilon \in \langle u \rangle L$ . Notons d'ailleurs que  $\varepsilon \in \langle u \rangle L$  ssi  $\langle u \rangle L \neq \emptyset$ .

**Question 11.** Pour chacune des égalités suivantes, dites lesquelles sont valides pour tous mots  $u$  et  $v$ , lettre  $a$ , et langages  $L, L_1, L_2$ . Justifiez vos réponses négatives par un contre-exemple.

- |  |  |
|--|--|
| (1) $\langle \varepsilon \rangle L = L$ ,                              | (2) $\langle a \rangle (L_1 \cdot L_2) = (\langle a \rangle L_1) \cdot L_2 \cup \langle a \rangle L_2$ , |
| (3) $\langle uv \rangle L = \langle u \rangle (\langle v \rangle L)$ , | (4) $\langle u \rangle (L^*) = (\langle u \rangle L) \cdot L^*$ .  |

**Question 12. a.** Programmez une fonction OCaml `eps_residu_ratexp : ratexp -> ratexp` qui à partir d'une expression rationnelle  $e$  construit une expression rationnelle  $e'$  telle que  $L(e') = \langle \varepsilon \rangle L(e)$ .

**b.** Donnez (et justifiez) un majorant, en fonction de  $|e|$ , de la taille  $|e'|$  de l'expression construite par votre programme.

**Question 13.** a. Programmez une fonction OCaml `char_residu_ratexp : char -> ratexp -> ratexp` qui, à partir de  $a \in A$  et  $e$ , construit une expression  $e''$  telle que  $L(e'') = \langle a \rangle L(e)$ .  
 b. Donnez (et justifiez) un majorant, en fonction de  $|e|$ , de la taille  $|e''|$  de l'expression construite par votre programme.

**Question 14.** a. Programmez une fonction OCaml `sousmot_de_ratexp : string -> ratexp -> bool` décidant si  $u \in \downarrow L(e)$  pour un mot  $u$  et une expression rationnelle  $e$ .  
 Indication : on pourra utiliser la fonction `char_residu_ratexp`.  
 b. Votre programme s'exécute-t-il en temps polynomial en  $|u| + |e|$ ? Justifiez brièvement votre réponse.

On développe maintenant une autre approche pour décider si  $u \in \downarrow L(e)$ . Pour un mot  $u = a_0 a_1 \cdots a_{n-1}$  et un langage  $L \subseteq A^*$ , on définit  $FC(u, L)$  comme étant l'ensemble des couples  $(i, j)$  tels que  $0 \leq i \leq j \leq |u|$  et  $u[i, j] \in L$ . Quand  $(i, j) \in FC(u, L)$  on dit que «  $L$  couvre le facteur  $[i, j]$  de  $u$  ». On écrit aussi  $FC(u, e)$  au lieu de  $FC(u, L(e))$  quand  $e$  est une expression rationnelle.

Pour représenter un ensemble de couples tel que  $FC(u, e)$ , on utilisera une matrice booléenne  $M$  de dimension  $(n + 1) \times (n + 1)$  telle que  $M[i, j] = \text{true}$  ssi  $(i, j) \in FC(u, e)$ . Notons qu'en particulier  $M[i, j] = \text{false}$  pour  $j < i$ .

**Question 15.** Programmez une fonction `facteurs_couverts : string -> ratexp -> bool array array` qui, pour un mot  $u$  et une expression  $e$ , calcule  $FC(u, e)$ . Indiquez et justifiez brièvement la complexité de votre fonction.  
 Pour ce code, il est suggéré de construire la matrice associée à une expression complexe  $e$  à partir des matrices associées aux sous-expressions de  $e$ .

**Question 16.** En utilisant la fonction `facteurs_couverts`, programmez une nouvelle version de la fonction `sousmot_de_ratexp` décidant si  $u \in \downarrow L(e)$  pour un mot  $u$  et une expression rationnelle  $e$  (cf. question 14). Indiquez la complexité de la nouvelle version.

\* \*  
 \*



