Sujet XENS 2019 : correction

Question 1. a. Supposons que $ua \preceq u'a'$. Notons n la taille de ua et n' celle de $u'a'$. Soit $p : ua \preceq u'a'$ un plongement de ua dans $u'a'$. Supposons que $a = a'$; on a $p(n-1) \leq n'-1$ donc, par stricte croissance de p, $p(i) < p(n-1) \leq n'-1$ pour tout $i < n-1$; la restriction de p à $[\![0, n-2]\!]$ est à valeurs dans $[\![0, n'-2]\!]$. La restriction de p à u est donc un plongement de u dans u'. On a donc $a = a'$ et $u \leq u'$. Supposons que $a \neq a'$. Alors p est un plongement de ua dans $u'a'$ tel que $p(n-1) \neq n'-1$. Ainsi p est à valeurs dans $[\![0, n'-2]\!]$; p est donc un plongement de ua dans u'.
Par conséquent, $ua \preceq u'$ ou ($a = a'$ et $u \leq u'$).
Réciproquement, supposons que $a = a'$ et $u \preceq u'$. Alors, en choisissant $p : [\![0, n-2]\!] \to [\![0, n'-2]\!]$ un plongement $u \preceq u'$, en posant $p(n-1) = n'-1$, on obtient un plongement $ua \preceq u'a'$.
Supposons que $ua \preceq u'$. Alors, en choisissant un plongement $p : [\![0, n-1]\!] \to [\![0, n'-2]\!]$ de ua dans u', on a, quitte à augmenter arbitrairement l'ensemble d'arrivée, un plongement $p : [\![0, n-1]\!] \to [\![0, n'-1]\!]$ qui montre que $ua \preceq u'a'$.
b.

```
let teste_sous_mot v u =
    let rec aux i j =
        if i>j then false
        else if i=0 then true
              else if v.[i-1]=u.[j-1] then aux (i-1) (j-1)
                    else aux i (j-1)
    in aux (String.length v) (String.length u);;
```

A chaque appel récursif, la taille du second argument diminue de un. Ainsi, la complexité est en $O(|u|)$.
Question 2.
a. On peut choisir $p(0) = 0$ ou $p(0) = 2$. Si $p(0) = 0$, on peut choisir $p(1) = 1$ ou $p(1) = 3$; si $p(0) = 2$, on peut choisir $p(1) = 3$. Il y a donc bien trois plongements : p_1, p_2, p_3 tels que $p_1(0) = 0$, $p_1(1) = 1$, $p_2(0) = 0$, $p_2(1) = 3$, $p_3(0) = 2$, $p_3(1) = 3$. Ainsi $\binom{abab}{ab} = 3$.
b. Toute injection croissante de $[m]$ dans $[n]$ est un plongement. Or il y a $\binom{n}{m}$ injections croissantes si $m \leq n$, 0 sinon. Il y a donc $\binom{n}{m}$ plongements si $m \leq n$, 0 sinon.
c. Soit p un plongement de va dans ua. Alors, en notant n la longueur de va et n' celle de ua, si $p(n-1) = n'-1$, la restriction de p à $[n-1]$ est une injection $[n-1] \to [n'-1]$ donc p réalise une injection de v dans u ; si $p(n-1) < n'-1$, p réalise une injection $[n] \to [n'-1]$ donc un plongement de va dans u.
Réciproquement, si $p : [n] \to [n'-1]$ est un plongement de va dans u, on obtient un plongement de va dans ua en considérant $p : [n] \to [n']$; si $p : [n-1] \to [n'-1]$ est un plongement de v dans u, en posant $p(n-1) = n'-1$, on obtient un plongement $p' : [n] \to [n']$ de va dans ua.
Ainsi, l'ensemble des plongements de va dans ua est (en bijection avec) la réunion disjointe des plongements de va dans u et des plongements de u dans v. On a donc $\binom{ua}{va} = \binom{u}{va} + \binom{u}{v}$.
Question 3.
a. La fonction auxiliaire est récursive ; elle renvoie un résultat pour $\min(i, j) = 0$; elle fait diminuer strictement l'élément $\min(i, j)$ à chaque appel récursif. Ainsi, elle renverra bien un résultat pour i et j des entiers naturels. Comme cette fonction auxiliaire est appelée pour des longueurs de chaînes de caractères, à savoir des entiers naturels, elle se terminera bel et bien.
b. Notons $f_{v,u}$ le résultat de la fonction. Si $v = \epsilon$, le nombre de plongements est 1, ce que renvoie bien la fonction : $f_{v,u} = \binom{u}{v}$; si $v \neq \epsilon$ et $u = \epsilon$, il n'y a pas de plongement et la fonction renvoie bien 0 : $f_{v,u} = \binom{u}{v}$; supposons que la fonction renvoie le bon résultat $\min(|u|, |v|) \leq n-1$, à savoir $f_{v,u} = \binom{u}{v}$; supposons que $\min(|u|, |v|) = n$; si les dernières lettres de v et u sont égales, à savoir $v = v'a$ et $u = u'a$, d'après la question précédente, $\binom{u}{v} = \binom{u'}{v'} + \binom{u'}{v}$ donc, par hypothèse de récurrence et par définition récursive de la fonction, $f_{v,u} = f_{v',u'} + f_{v,u'} = \binom{u'}{v'} + \binom{u'}{v} = \binom{u}{v}$. Si les dernières lettres de v et u ne sont pas égales et $v = v'a$; $u = u'a'$ avec $a \neq a'$, alors $\binom{u}{v} = \binom{u'}{v}$; or la fonction renvoie $f_{v,u} = f_{v,u'}$ et $f_{v,u'} = \binom{u'}{v}$ par hypothèse de récurrence ; par conséquent, là encore, $f_{v,u} = \binom{u}{v}$.
Ceci prouve par récurrence sur $\min(|u|, |v|)$ que pour tout u, v, $f_{v,u} = \binom{u}{v}$, à savoir la fonction renvoie bien le nombre de plongements de v dans u.
Question 4.
a. Soit $C_{p,n}$ la complexité de la fonction auxiliaire "aux p n". On a, par définition de la fonction, $C_{p,n} \leq 2 + C_{p-1,n-1} + C_{p,n-1}$ et $C_{n,p} \leq 1 + C_{p,n-1}$.
En outre, si $n = 0$, pour tout p, $C_{p,n} = 1 < 2 \times 2^n$. Soit $n \geq 1$. Supposons que $C_{p,n-1} < 2^n$ pour tout entier p. Alors, pour tout entier p, $C_{p,n} \leq 2 + C_{p-1,n-1} + C_{p,n-1} < 2 + 2^n + 2^n = 2 + 2^{n+1} < 2^{n+1}$.

Ainsi, par récurrence, $C_{p,n} < 2 \times 2^n$ pour tout p et n entiers naturels. Or $T(v,u) \leq C_{|v|,|u|} < 2 \times 2^{|u|}$.
b. On peut très bien n'avoir aucun plongement de v dans u. Pour autant il faut toujours faire au moins $|u|$ appels à la fonction aux.
On peut aussi choisir $u = v$. Dans ce cas, il y aura aussi au moins $n = |u|$ appels récursifs alors qu'il n'y a qu'un seul plongement.
c. Soit $C_{i,j}$ le nombre d'appels à la fonction aux pour le calcul de "aux i j". Montrons par récurrence que pour tout i, j, $C_{i,j} \geq 2\binom{u[0,j[}{v[0,i[} - 2$.
Pour tout i, $C_{i,0} = 1$ et $2\binom{u[0,0[}{v[0,i[} - 2 = 2 - 2 = 0$. On a donc pour tout i, $C_{i,0} \geq 2\binom{u[0,0[}{v[0,i[} - 2$.
Soit $j \geq 1$ fixé. Supposons que pour tout i, $C_{i,j-1} \geq 2\binom{u[0,j-1[}{v[0,i[} - 2$.
Supposons que $v[i-1] = u[j-1]$. Alors $C_{i,j} = 2 + C_{i-1,j-1} + C_{i,j-1} \geq 2 + 2\binom{u[0,j-1[}{v[0,i-1[} - 2 + 2\binom{u[0,j-1[}{v[0,i-1[} - 2 = 2(\binom{u[0,j-1[}{v[0,i-1[} + \binom{u[0,j-1[}{v[0,i[}) - 2 = 2\binom{u[0,j[}{v[0,i[} - 2$.
Supposons que $v[i-1] \neq u[j-1]$. Alors $C_{i,j} = 1 + C_{i,j-1} \geq 1 + 2\binom{u[0,j-1[}{v[0,i[} - 2 = 2\binom{u[0,j[}{v[0,i[} - 1 \geq 2\binom{u[0,j[}{v[0,i[} - 2$.
Ceci démontre le résultat par récurrence. Comme $T(v,u) = 1 + C_{|v|,|u|}$, $T(v,u) \geq 1 + 2\binom{u}{v} - 2 = 2\binom{u}{v} - 1$. **Question 5.**

```
let nb_plongements_rapide v u =
    let n=String.length u and p=String.length v in
    let t=Array.make_matrix p n -1 in
    for j=0 to n do
        t.(0).(j)<-1
        done;
    for i=0 to p do
        t.(i).(0)<-0;
    let rec aux i j=
        if t.(i).(j)>-1 then t.(i).(j)
        else
        begin
        if u.[i-1]=v.[j-1] then
            begin
            t.(i).(j)<-aux (i-1) (j-1)+aux i (j-1);
            t.(i).(j)
            end
        else
            begin
            t.(i).(j)<-aux i (j-1);
            t.(i).(j)
            end
        end in
     aux p n;;
```

La complexité spatiale est en $O(np)$ pour stocker le tableau. Pour chaque case, une fois que sa valeur a été modifiée, elle ne créera plus d'appel récursif. Ainsi, il ne peut y avoir que $|u \times |v|$ appels récursifs.
Question 6.
a. Soit u un sous-mot de $wava$. Si u ne se termine pas par a, u est un sous-mot de wav. Si u se termine par un a, alors $u = u'a$. Si u' est un sous-mot de w, u est un sous-mot de wa donc de wav ; sinon u' est un sous-mot de wav sans être un sous-mot de w. Ainsi, $\downarrow wava \subset \downarrow wav \cup (\downarrow wav \backslash \downarrow w).a$.
Réciproquement, soit u un sous-mot de wav. Alors c'est un sous-mot de $wava$. Soit u' un sous-mot de wav qui n'est pas un sous-mot de w. Alors $u'a$ est un sous-mot de $wava$. On a donc $\downarrow wava \supset \downarrow wav \cup (\downarrow wav \backslash \downarrow w).a$.
En conclusion, $\downarrow wava = \downarrow wav \cup (\downarrow wav \backslash \downarrow w).a$.
b. Supposons que v ne contient pas par a. Soit u un sous-mot de wav. Alors, si u se termine par a, comme le suffixe v de wav n'a pas de a, u est un sous-mot de wa donc $u = u'a$ où u' est un sous-mot de w. Ainsi, si $u \in \downarrow wav \cap (\downarrow wav).a$, alors $u \in \downarrow w$. Par conséquent, $\downarrow wav \cap (\downarrow wav \backslash \downarrow w).a = \emptyset$.
Réciproquement, supposons que v contient un a. Notons v_i cette lettre dans $v = v_0 \cdots v_{n-1}$. Alors $wav_0 \cdots v_i$ est un sous-mot de wav. De plus, il s'écrit $wav_0 \cdots v_{i-1}.a$; $wav_0 \cdots v_{i-1}$ n'est pas un sous-mot de w (il est de taille strictement supérieure). Par conséquent, $wav_0 \cdots v_{i-1}$ est un sous-mot de wav qui n'est pas un sous-mot de w.
En conclusion, $wav_0 \cdots v_{i-1}v_i \in \downarrow wav \cap (\downarrow wav \backslash \downarrow w).a$. Ainsi, ces deux ensembles ne sont pas disjoints.
On a donc bien $\downarrow wav \cup (\downarrow wav \backslash \downarrow w).a$ disjointe si et seulement si v ne contient aucun a.
Question 7.
a. Supposons que $u = u'a$.
Si u' ne contient pas la lettre a, alors les sous-mots de u sont les sous-mots v de u' et les mots va où v est un sous-mot de u'. L'union de ces deux types de sous-mots est encore disjointe ; de plus $\mathrm{Card}((\downarrow u').a) = \mathrm{Card}(\downarrow u')$. On a donc $\mathrm{Card}(\downarrow u) = \mathrm{Card}(\downarrow u') + \mathrm{Card}((\downarrow u').a) = 2\,\mathrm{Card}(\downarrow u')$.

Si u' contient la lettre a, soit w et v les mots tels que $u = wava$ et v ne contient pas a (v peut-être vide). On a alors $\mathrm{Card}((\downarrow wav \backslash w).a) = \mathrm{Card}(\downarrow wav) - \mathrm{Card}(\downarrow w)$. D'après la question précédente, $\mathrm{Card}(\downarrow u) = \mathrm{Card}(\downarrow wav) + \mathrm{Card}((\downarrow wav \backslash \downarrow w).a) = 2\,\mathrm{Card}(\downarrow wav) - \mathrm{Card}(\downarrow w)$.
b.

```
let nb_sousmots u=
    let n=String.length u in
    let l=Array.make (n+1) -1 in
    l.(0)<-1;
    let rec aux i =
        if l.(i)=-1 then
        begin
        let a=u.[i-1] in
        let j=ref i-2 in
            while j>0 and not !b do
                if u.[j]=a then b:=true
                else j:=!j-1
            done;
        if not !b then l.(i)<-2*aux (i-1)
        else l.(i)<-2*l.(i-1)-l.(!j-1)
        end;
        l.(i) in
    aux n;;
```

Chaque case du tableau est remplie au moins une fois donc effectue un appel récursif. Pour remplir chaque case du tableau, il faut au pire parcourir tout le mot puis effectuer au plus deux opérations arithmétiques. Chaque remplissage est donc en $O(|u|)$ opérations. Comme il y a $|u|+1$ cases à remplir, la complexité est en $O(|u|^2)$. La complexité est donc bien polynômiale en $|u|$.

Question 8.

a. Par définition, ua et vb sont des sous-mots de wa. Ainsi, u est un sous-mot de w ; comme $a \neq b$, vb est un sous-mot de w. Ains u est un sous-mot de w et vb est un sous-mot de w. Si w' est un sur-mot de u et de vb, alors $w'a$ est un sur-mot de ua et de vb. Ainsi $w'a$ est de longueur supérieur à wa donc w' est de longueur supérieur à w. De plus, si w' et w ont même longueur, $w'a$ et wa aussi donc wa est inférieur à $w'a$ pour l'ordre lexicographique ; par définition de celui-ci, w est inférieur à w' dans l'ordre lexicographique. En conclusion, w est le sur-mot de u et vb de longueur minimale et, parmi ceux de longueur minimale, le plus petit dans l'ordre lexicographique. Ceci signifie que $w = pcsmc(u, vb)$.
b. Supposons que $a \neq b$. Si w est un sur-mot de ua et vb, si sa dernière lettre est c avec $c \neq a$ et $c \neq b$, $w = w'c$, d'après la question initiale, w' est un sur-mot de ua et vb. Ainsi, w n'est pas de longueur minimale. Ainsi, $pcsmc(ua, vb)$ se termine par a ou b. D'après la question précédente, si $pcsmc(ua, vb) = wa$, alors $w = pcsmc(u, vb)$. Par symétrie, si $pcsmc(ua, vb) = wb$, alors $pcsmc(ua, vb) = pcsmc(vb, ua) = wb$ donc $w = pcsmc(v, ua)$.
On a donc $pcsmc(ua, vb) = pcsmc(u, vb)a$ ou $pcsmc(ua, vb) = pcsmc(ua, v)b$. Ainsi, $pcsmc(ua, vb)$ est le mot de longueur inférieure parmi $pcsmc(u, vb)a$ et $pcsmc(ua, v)b$ ou le plus petit des deux dans l'ordre lexicographique.
Si $a = b$, si w est un sur-mot de ua et va, alors, si $w = w'c$ ne se termine pas par a, alors w' est un sur-mot de ua et va donc w n'est pas de longueur minimale. Ainsi, $pcsmc(ua, va) = wa$; en outre, w est un sur-mot de u et de v ; en outre, si w' est un sur-mot de u et de v, $w'a$ est un sur-mot de ua et va donc $w'a$ est de longueur supérieure à wa donc w' est de longueur supérieure à w. En outre, si $|w'| = |wl$, alors w' est supérieure à w dans l'ordre lexicographique. Par conséquent, $w = pcsmc(u, v)$.
En conclusion, si $u = \epsilon$, $pcsmc(u, v) = v$; si $a \neq b$, $pcsmc(ua, vb)$ est le plus petit en longueur ou dans l'ordre lexicographique parmi $pcsmc(u, vb)a$ et $pcsmc(ua, v)a$; enfin $pcsmc(ua, va) = pcsmc(u, v)a$.
c.

```
let pcsmc u v=
    let n=String.length u and p=String.length v in
    let t=Array.make_matrix (n+1) (p+1) "" in
    for i=0 to n do
        t.(i).(0)<-String.sub u 0 (i-1)
    done;
    for j=0 to p do
        t.(0).(j)<-String.sub v 0 (j-1)
    done;
    let rec aux i j =
        if i=0 && j=0 then ""
        else if t.(i).(j)<>"" then t.(i).(j)
        else begin
             if u.[i-1]=v.[j-1]
```

```
            then t.(i).(j)<-t.(i-1).(j-1)^v.[j-1]
            else
                let w1=t.(i-1).(j)^u.[i-1] and w2=t.(i).(j-1)^v.[j-1] in
                let n1=String.length w1 and n2=String.length w2 in
                if n1<n2 then t.(i).(j)<-w1
                else if n2<n1 then t.(i).(j)<-w2
                else if w1<w2 then t.(i).(j)<-w1
                else t.(i).(j)<-w2;
            t.(i).(j) in
    aux n p;;
```

La complexité spatiale est en $O(np)$ où $n = |u|$ et $p = |v|$. Chaque case est calculé au plus une fois. Pour chaque case, le calcul effectué est en $O(1)$ opérations à partir des cases déjà remplies. Ainsi, la complexité globale est en $O(np)$.

Question 9.

$e_1 = \big(a + \emptyset)c\big)^*.\big(b(\emptyset.(cc)^*)\big)$. Ce langage étant vide, un mot de $L(e_1)$ ne peut commencer par a, par b ni par c.
$e_2 = (ba)^*\big((\epsilon + a)c^*\big)$. Unt mot de $L(e_2)$ peut donc commencer par b, par a ou par c.

Question 10.

```
let rec cont_epsilon e = match e with
    | Empty -> false
    | Epsilon -> true
    | Letter _ -> false
    | Sum(e1,e2) -> cont_epsilon e1 || cont_epsilon e2
    | Product(e1,e2) -> cont_epsilon e1 && cont_epsilon e2
    | Star(_) -> true;;

let rec est_vide e = match e with
    | Empty -> true
    | Epsilon -> false
    | Letter _ -> false
    | Sum(e1,e2) -> est_vide e1 && est_vide e2
    | Product(e1,e2) -> est_vide e1 || est_vide e2
    | Star(_) -> false;;

let rec peut_debuter_par e a = match e with
    | Empty -> false
    | Epsilon -> false
    | Letter b -> b=a
    | Sum(e1,e2) -> peut_debuter_par e1 a || peut_debuter_par e2
    | Product(e1,e2) ->
    (peut_debuter_par e1 a && not(est_vide e2))||(cont_epsilon e1 && peut_debuter_par e2 a)
    | Star(e1) -> peut_debuter_par e1 a;;
```

Question 11.

(1) Supposons que $L = \{aa\}$. Soit $v = a$. Alors $w = a$ est un sur-mot de ϵ et $wv = aa \in L$ donc $a \in < \epsilon > L$ mais $a \notin L$. L'égalité est donc fausse.
(2) Supposons que $L_1 = \emptyset$. Alors $L_1L_2 = \emptyset$ donc $< a > (L_1L_2) = \emptyset$. Supposons que $L_2 = \{a\}$. Alors $\epsilon \in < a > L_2 \neq \emptyset$. L'égalité est donc fausse.
(3) Supposons que $L = \{ab\}$, $u = b$, $v = a$. Alors uv n'est un sous-mot d'aucun mot de L donc $< uv > L = \emptyset$. De plus, $< v > L = < a > \{ab\} = \{b, \epsilon\}$ donc $< u > (< v > L) = < b > \{\epsilon, b\} = \{\epsilon\}$. Ainsi, l'égalité est fausse.
(4) Si $L = \emptyset$, $u = \epsilon$, $L^* = \{\epsilon\}$, $< u > L^* = \{\epsilon\}$. Or $< \epsilon > \emptyset = \emptyset$ donc $(< u > L)L^* = \emptyset.\{\epsilon\} = \emptyset$. L'égalité est donc fausse.

Question 12.

a.On remarque que $< \epsilon > L$ est l'ensemble des suffixes de L.

```
let rec eps_residu_ratexp e =
    | Empty -> Empty
    | Epsilon -> Epsilon
    | Letter _ -> Sum(Epsilon,e)
    | Sum(e1,e2) -> Sum(eps_residu_ratexp e1,eps_residu_ratex e2)
    | Product(e1,e2) ->
    if est_vide e1 then Empty
    else if cont_epsilon e2 then Sum(eps_residu_ratexp e1,Sum(eps_residu_ratexp e2,Product(eps_residu_ratexp e1,e2)))
    else Sum(eps_residu e2,Product(eps_residu_ratexp e1,e2))
    | Star(e1) ->
```

```
    if est_vide e1 then Epsilon
    else Product(eps_residu_ratexp e1,Star(e1));;
```

b. Notons C_n la taille dans le pire des cas d' une expression e' pour $|e| = n$. On a $C_n \leq 2C_{n_1} + C_{n_2} + O(n)$ où $n_1 + n_2 = n$. Supposons pour simplifier que $n_1 = n_2 = n/2$ et $n = 2^p$. Alors $C_{2^p} \leq 2C_{2^{p-1}} + C_{2^{p-1}} + O(2^p)$. On a donc $\frac{C_{2^p}}{2^p} \leq \frac{3}{2}\frac{C_{2^{p-1}}}{2^{p-1}} + O(1)$. Ainsi, $\frac{C_{2^p}}{2^p} = O((\frac{3}{2})^p)$ donc $C_{2^p} = O(3^p) = O((2^p)^{\ln_2(3)}) = O((2^p)^2)$. Ainsi, $C_n = O(n^2)$.
La complexité n'est donc pas pire que quadratique.

Question 13.

a.

```
let rec char_residu_ratexp a e = match e with
    | Empty -> Empty
    | Epsilon -> Empty
    | Letter b -> if a=b then Epsilon else Empty
    | Sum(e1,e2) -> Sum(char_residu_ratexp a e1,char_residu_ratexp a e2)
    | Product(e1,e2) ->
    if est_vide e1 then Empty
    else Sum(Product(char_residu_ratexp a e1,e2),char_residu_ratexp e e2)
    | Star(e1) ->
    if est_vide e1 then Empty
    else Product(char_residu_ratexp a e1,e1);;
```

b. Avec les notations de la question 12, $C_n \leq 2C_{n/2} + O(n)$. Ceci donne une complexité en $C_n = O(n \ln(n))$.

Question 14.

a. Si $u = au'$, $v \in< u > L$ si et seulement s'il existe $au' = u \preceq w$ tel que $wv \in L$, c'est-à-dire qu'il existe w_1 qui contient a, $u' \preceq w_2$ tel que $w_1 w_2 v \in L$. Ceci équivaut à ce qu'il existe $u' \preceq w_2$ tel que $w_2 v \in< a > L$, ce qui équivaut à $v \in< u' > (< a > L)$.

```
let rec sousmot_de_ratexp u e =
    if u="" then not (est_vide (eps_residu_ratexp e))
    else sousmot_ratexp (String.sub u 1 (String.length u -1)) (char_residu_ratexp u.[0] e);;
```

b. Pour chaque lettre de u, on fait appel à "char_residu_ratexp" qui s'exécute en temps (moins que) polynômiale ; on fait appel à aussi à "eps_residu_ratexp" à la fin, qui est en temps quadratique donc polynômiale. Au total, on a donc une complexité polynômiale (et même en $O(|u| \times |e| \ln |e| + |e|^2)$).

Question 15.

```
let somme_mat_bool m1 m2 =
    let n=Array.length m1 in
    let m=Array.make n n false in
    for i=0 to n-1 do
        for j=0 to n-1 do
            m.(i).(j)<-m1.(i).(j) || m2.(i).(j)
        done
    done;
    m;;

let prod_mat_bool m1 m2=
    let n=Array.length m1 in
    let m=Array.make n n false in
    for i=0 to n-1 do
        for j=0 to n-1 do
            for k=0 to n-1 do
                m.(i).(j)<-m.(i).(j) || (m1.(i).(k) && m2.(k).(j))
            done
        done
    done;
    m;;

let mat_id_bool n =
    let m=Array.make_matrix n n false in
    for i=0 to n-1 do
        m.(i).(i)<-true
    done;
    m;;
```

```
let mat_etoile_bool m =
    let n=Array.length m in
    let m0=ref Array.make_matrix n n false and m1=ref Array.make_matrix n n false in
    for i=0 to n-1 do
        for j=0 to n-1 do
            m0.(i).(j)<-m.(i).(j)
            m1.(i).(j)<-m.(i).(j)
        done
    done;
    for k=2 to n do
        m1:=prod_mat_bool(!m1,m);
        m0:=somme_mat_bool(!m0,!m1)
    done;
    !m0;;

let rec facteurs_couverts u e =
    match e with
        | Empty ->
        let n=String.length u in Array.make_matrix (n+1) (n+1) false
        | Epsilon -> mat_id_bool (String.length u +1)
        | Letter a ->
        let n=String.length u in let m=mat_id_bool (n+1) in
        for i=0 to n-1 do
            if u.[i]=a then m.(i).(i+1) <- true
        done;
        m
        | Sum(e1,e2) -> somme_mat_bool (facteurs_couverts u e1) (facteurs_couverts u e2)
        | Product(e1,e2) -> prod_mat_bool (facteurs_couverts u e1) (facteurs_couverts u e2)
        | Star(e1) -> mat_etoile_bool (facteurs_couverts u e1);;
```

Notons $n = |u|$. La complexité de la fonction "somme" est $O(n^2)$, celle de la fonction produit est $O(n^3)$., celle de la fonction "etoile" est $O(n^4)$. Au total, la complexité pour une expression de taille p vérifie $C_{2^p} = 2C_{2^{p-1}} + O(n^4)$. Cela donne une complexité polynômiale en n et 2^p, plus précisément en $O(pn^4)$. Pour une expression de taille p, la complexité est donc en $O(\ln(p)n^4)$.

Question 16.

```
let sous_mot_de_ratexp u e = (facteurs_couverts u e).(0).(String.length u);;
```