DS 2-C

X-ENS 1993 sous-mots

MPI/MPI*, lycée Faidherbe

- Un mot est une suite de lettres $a_0 \dots a_{n-1}$ tirées d'un alphabet fini $A = \{a, b, \dots\}$.
  On utilisera $u, v, u', u'', u_1, u_2, \dots$ pour dénoter les éléments de A^*.
  On note ϵ pour le mot vide et $|u|$ pour la longueur de u, de sorte que $|\epsilon| = 0$.
- 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 $u_1 = \epsilon$ (ou si $u_2 = \epsilon$) dans cette décomposition.
  Dans le cas d'un mot $u = a_0 \dots a_{n-1}$, on écrit « $u[i, j[$ », sous la condition $0 \leqslant i \leqslant j \leqslant n$, pour désigner le facteur $a_i, \dots a_{j-1}$.
  Cette notation s'étend à $u[i \dots [$ 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 $u = a_0 \dots a_{n-1}$, on dira qu'un mot v de longueur m est un *sous-mot* de u, ce que l'on notera $v \preccurlyeq u$, s'il existe une suite strictement croissante $0 \leqslant p_0 < p_1 < \dots < p_{m-1} < n$ telle que $v = a_{p_0} a_{p_1} \dots a_{p_{m-1}}$. Par exemple, *caml* $\preccurlyeq$ *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 $p : [m] \to [n]$. 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 $p : v \preccurlyeq u$.
  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 $k_0, \dots, k_{r-1}$, on dit que A a une complexité en $O\big(f(k_0, \dots, k_{r-1})\big)$ s'il existe une constante $C > 0$ telle que, pour

toutes les valeurs de $k_0, \dots, k_{r-1}$ suffisamment grandes (c'est-à-dire plus grandes qu'un certain seuil), pour toute instance du problème de paramètres $k_0, \dots, k_{r-1}$, la complexité est au plus $C \cdot f(k_0, \dots, k_{r-1})$.

OCaml

On rappelle quelques éléments du langage OCaml qui peuvent être utiles. Une chaine de caractères `s` a le type `string`, sa longueur est obtenue avec `String.lenght s` et son `i`-ième caractère avec `s.[i]`, les caractères étant indexés à partir de 0. Un tableau `t` a le type τ `array`, où τ 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/la candidat.e est libre d'utiliser tout autre élément du langage OCaml et de sa bibliothèque standard.

Question 1

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

$$ua \preccurlyeq u'a' \iff ua \preccurlyeq u' \text{ ou } (a = a' \text{ et } u \preccurlyeq u')$$

2. 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.

1.

- *Si $ua = v_0v_1 \cdots v_{m-1}$ est un sous mot de $u'a' = v'_0v'_1 \cdots v'_{n-1}$, on note p une application strictement croissante de $[m]$ vers $[n]$ telle que $v'_{p(k)} = v_k$.*
  - *Si $p(m-1) \neq n-1$ alors $p(m-1) < n-1$ donc p est qui définit ua comme un sous-mot de $v'_0v'_1 \cdots v'_{n-1} = u'$.*
  - *Si $p(m-1) \neq n-1$ alors la restriction de p à $[m-1]$ est une application strictement croissante vers $[n-1]$, elle définit u comme sous-mot de u'. De plus $a' = v'_{n-1} = v'_{p(m-1)} = v_{m-1} = a$.*

  Notons qu'on a en fait prouvé $ua \preccurlyeq u'a' \Rightarrow (a \neq a' \text{ et } ua \preccurlyeq u')$ ou $(a = a' \text{ et } u \preccurlyeq u')$.
- *On suppose maintenant qu'on a $ua \preccurlyeq u'$ ou $u \preccurlyeq u'$ avec $a = a'$.*
  - *Si $ua \preccurlyeq u'$ alors, un plongement de ua dans u' peut être considéré comme un plongement de ua dans $u'a'$ donc $ua \preccurlyeq u'a'$.*
  - *Si $u \preccurlyeq u'$ et $a = a'$ alors un plongement de u dans u' peut se prolonger en un plongement de ua dans $u'a$ en envoyant le dernier a de ua dans a en dernière position de $u'a$: on a bien $ua \preccurlyeq u'a = u'a'$.*

  Dans les deux cas on a $ua \preccurlyeq u'a'$.

Comme on a $(a \neq a' \text{ et } ua \preccurlyeq u') \Rightarrow (ua \preccurlyeq u')$, le résultat prouvé est plus fort et c'est celui qu'on utilisera par la suite :

$$ua \preccurlyeq u'a' \iff (a \neq a' \text{ et } ua \preccurlyeq u') \text{ ou } (a = a' \text{ et } u \preccurlyeq u')$$

2. Pour séparer récursivement la dernière lettre du reste d'une chaîne, on travaille sur les indices.

```
let teste_sous_mot v u =
  let rec aux i j =
    match i, j with
    | 0, _ -> true  (* v  est vide *)
    | _, 0 -> false (* u est vide mais pas v *)
    | _ when v.[i - 1] = u.[j - 1] -> aux (j - 1) (i - 1)
    | _  -> aux j (i - 1)
  in aux (String.length v) (String.length u)
```

Chaque appel récursif à `aux` *se fait en temps constant.*
À chaque appel i décroît de 1 donc le nombre d'appels est majoré par $|u|$.
La complexité de la fonction est en $\mathcal{O}(|u|)$.

I Compter les plongements

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

Question 2

1. Montrez que $\binom{abab}{ab} = 3$.
2. Que vaut $\binom{a^n}{a^m}$ quand $a \in A$ est une lettre ?
3. Déterminer $\binom{ua'}{va}$ en fonction de $\binom{u}{va}$ et $\binom{u}{v}$ pour tous mots u, v et toutes lettres $a, a' \in A$. On séparera les cas $a = a'$ et $a \neq a'$.

---

1. *Il y a 3 plongement de ab dans $abab$: $\underline{ab}ab$, $\underline{a}b\underline{ab}$ et $ab\underline{ab}$.*
2. *Pour plonger a^m dans a^n on doit sélectionner m positions dans l'ensemble $[n]$ à n éléments : il y a $\binom{n}{m}$ choix d'où $\binom{a^n}{a^m} = \binom{n}{m}$.*
3. *On note $m = |u|$, $n = |u|$ et p une application strictement croissante de $[m+1]$ vers $[n+1]$ qui définit un plongement de $va = v'$ dans $ua' = u'$.*
   - *Si $a \neq a'$ alors $p(m)$ ne peut être égal à n car $u'_{p(m)} = v'_m = a \neq a' = u'_n$.*
     Comme p est strictement croissante elle est donc une application strictement croissante de $[m+1]$ vers $[n]$: p définit un plongement de va dans u. Inversement tout plongement de va dans u définit un plongement de va dans ua'.
     Ainsi on a $\binom{ua'}{va} = \binom{u}{va}$.
   - *Si $a = a'$ alors les plongement de va dans ua peuvent être séparés en deux ensembles distincts.*
     — *Si $p(m) = n$ alors la restriction de p à $[m]$ définit bijectivement un plongement de v dans u : il a ainsi $\binom{u}{v}$ plongements de ce type.*
     — *Si $p(m) \neq n$ alors p est défini par un plongement de va dans u : il en a $\binom{u}{va}$.*
     On a donc $\binom{ua'}{va} = \binom{u}{va} + \binom{u}{v}$

Pour calculer $\begin{pmatrix} u \\ v \end{pmatrix}$ on considère la fonction OCaml suivante.

```
let nb_plongements v u =
  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)
```

Question 3

1. Prouvez sa terminaison.
2. Justifiez sa correction, c'est-à-dire, expliquez pourquoi elle renvoie bien la valeur $\begin{pmatrix} u \\ v \end{pmatrix}$.

---

1. *i et j sont des entiers. La valeur de i décroît et celle de j décroît strictement lors de chaque appel. Comme la fonction termine lorsque i ou j est nul, on endéduit que i et j garde des valeurs positives. Ainsi $i+j$ est positif, strictement décroissant donc, par récurrence sur la valeur de $i+j$, la fonction termine.*
2. *La correction de* `aux` *se prouve aussi par récurrence sur $i+j$:*

$$\texttt{aux i j}\ \textit{calcule}\ \begin{pmatrix} u_0 \cdots u_{j-2} u_{j-1} \\ v_0 \cdots v_{i-2} v_{i-1} \end{pmatrix}$$

Le cas $i+j=0$ donne $i=j=0$ et renvoie 1 qui est bien $\begin{pmatrix} \varepsilon \\ \varepsilon \end{pmatrix}$.

Pour $i+j>0$ on sépare en 4 cas :

(a) $i=0$ donc on calcule $\begin{pmatrix} u_0 \cdots u_{j-1} \\ \varepsilon \end{pmatrix}$ qui vaut bien 1.

(b) $i>0$ et $j=0$ donc on calcule $\begin{pmatrix} \varepsilon \\ v_0 \cdots v_{i-1} \end{pmatrix}$ qui vaut bien 0.

(c) $i>0$ et $j>0$ avec $u_{j-1}=v_{i-1}=a$; on veut calculer $\begin{pmatrix} u_0 \cdots u_{j-2} a \\ v_0 \cdots v_{i-2} a \end{pmatrix}$ dont on a vu qu'il était égal à $\begin{pmatrix} u_0 \cdots u_{j-2} \\ v_0 \cdots v_{i-2} a \end{pmatrix} + \begin{pmatrix} u_0 \cdots u_{j-2} \\ v_0 \cdots v_{i-2} \end{pmatrix} = \begin{pmatrix} u_0 \cdots u_{j-2} \\ v_0 \cdots v_{i-2} v_{i-1} \end{pmatrix} + \begin{pmatrix} u_0 \cdots u_{j-2} \\ v_0 \cdots v_{i-2} \end{pmatrix}$.
D'après l'hypothèse de récurrence cette dernière somme est égale à `aux i (j-1)` *+*`aux (i-1) (j-1)` *ce qui est bien le résultat renvoyé.*

(d) De même pour le dernier cas.

On note $T(v,u)$ le nombre d'appels de la fonction `aux` lors du calcul de `nb_plongements v u`.

Question 4

1. Montrez qu'il existe une constante C_1 telle que $T(v,u) < C_1.2^{|u|}$.
2. Montrez que l'on ne peut pas majorer $T(v,u)$ par une fonction polynomiale de $\binom{u}{v}$.
3. Montrez qu'il existe une constante C_2 telle que $T(v,u) \geqslant 2\binom{u}{v} + C_2$.

---

1. *On note $T'(i,j)$ le nombre de fois où la fonction* `aux` *est appelée lors du calcul de* `aux i j`.
   On a $T(v,u) = T'(|v|,|u|)+1$.
   De plus $T'(0,j) = 0$, $T'(i,0) = 0$, $T'(i,j) \in \{2+T'(i-1,j-1)+T'(i,j-1),1)+T'(i,j-1)\}$.
   Si on note u_j un majorant des $T'(i,j)$ pour $i \in \mathbb{N}$, on a $u_0 = 0$ et $u_j \leqslant 2+2u_{j-1}$.
   On a alors $2^{-j}.u_j \leqslant 2^{-(j-1)}.u_{j-1} + 2^{-(j-1)}$ donc $2^{-j}.u_j \leqslant 2^{-0}.u_0 + 2^{-(j-1)} + \cdots + 2^{-0}$ puis $u_j \leqslant 0+2+4+\cdots+2^j = 2^{j+1}-2$.
   On trouve donc $T(v,u) \leqslant 2^{|u|+1} - 1 \leqslant 2.2^{|u|}$; on peut choisir $C_1 = 2$.
2. *Pour $u = a^n$ et $v = ba^m$ avec $n < m$ on a $T'(i,j) = 2 + T'(i-1,j-1) + T'(i,j-1)$ pour tout $j > 1$ car on a toujours $i > j$ doncles lettres considérées sont toujours des a.*
   On prouve alors, par récurrence sur j, que $T'(i,j) = 2^{j+1} - 2$ donc $T(v,u) = 2^{|u|+1} - 1$.
   Or $\binom{u}{v} = 0$: donc il n'est pas possible de majorer $T(v,u)$ par une fonction polynomiale de $\binom{u}{v}$ (et même par n'importe quelle fonction de $\binom{u}{v}$) puisque $T(v,u)$ peut prendre des valeurs arbitrairement grandes.
3. *On a $T'(0,n) = 0$ donc $T(\varepsilon,u) = 1$ alors que $\binom{u}{\varepsilon} = 1$: $T(\varepsilon,u) = 2\binom{u}{\varepsilon} - 1$.*
   On va prouver qu'on a $T(v,u) \geqslant 2\binom{u}{v} - 1$.
   Pour cela on prouve, par récurrence sur $i+j$, que $T'(i,j) \geqslant 2B(i,j) - 2$ avec $B(i,j) = u_0 \cdots u_{j-2}u_{j-1}v_0 \cdots v_{i-2}v_{i-1}$.
   (a) *On a vu qu'il y avait égalité pour $i = 0$.*
   (b) *Pour $j = 0$ (et $i > 0$) on a $B(i,0) = 0$ et $B'(i,0) = 0 \geqslant 0 - 2$.*
   (c) *Pour $i > 0$ et $j > 0$ avec $u_{i-1} = u_{j-1}$ on a*
   $T'(i,j) = 2 + T'(i-1,j-1) + T'(i,j-1) \geqslant 2 + 2B(i-1,j-1) - 2 + 2B(i,j-1) - 2$
   donc $T'(i,j) \geqslant 2\big(B(i-1,j-1) + B(i,j-1)\big) - 2$.
   Or la question **4.** *montre qu'on a, dans ce cas, $B(i-1,j-1) + B(i,j-1) = B(i,j)$: on aboutit à $T'(i,j) \geqslant 2B(i,j) - 2$*
   (d) *Pour $i > 0$ et $j > 0$ avec $u_{i-1} \neq u_{j-1}$ on a*
   $T'(i,j) = 1 + T'(i,j-1) \geqslant 1 + 2B(i,j-1) - 2 \geqslant 2B(i,j-1) - 2$.
   La question **4.** *donne $B(i,j-1) = B(i,j)$ d'où $T'(i,j) \geqslant 2B(i,j) - 2$*
   La propriété est démontrée : on peut choisir $C_2 = -1$.

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 conserver des valeurs calculées dans un tableau.

Le problème est que l'on calcule les mêmes valeurs plusieurs fois, le principe de la programmation dynamique est de ne calculer qu'une fois les valeurs en les stockant dans une matrice car il y a deux indices.

```
let nb_plongements_rapide v u =
  let n = String.length u
  and m = String.length v in
  let aux = Array.make_matrix (m+1) (n+1) 0 in
  (* Si v = epsilon, (v parmi u) = 1 *)
  for j = 0 to n do aux.(0).(j) <- 1 done ;
  (* Si u = epsilon, (v parmi u) = 0 déjà marqué *)
  for j = 1 to m do
    for i = 1 to n do
      if v.[i - 1] = u.[j - 1]
      then aux.(i).(j) <- aux.(i).(j-1) + aux.(i-1).(j-1)
      else aux.(i).(j) <- aux.(i).(j-1) done done;
  aux.(m).(n)
```

On crée une matrice de taille $(|v|+1)\times(|u|+1)$: la complexité spatiale est un $\mathcal{O}\big(|u|\times|v|\big)$. On fait un nombre fini d'opérations dans deux boucles imbriquées donc la complexité temporelle est aussi un $\mathcal{O}\big(|u|\times|v|\big)$. Les deux complexités sont donc quadratiques en $|u|+|v|$.

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

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

Question 6

1. Montrez que, pour tous mots v, w et toute lettre a, on a :

   $$(\ddagger) \qquad (\downarrow wava) = (\downarrow wav) \cup \big((\downarrow wav) \setminus (\downarrow w)\big) \cdot a$$

2. Montrez que les langages $(\downarrow wav)$ et $\big((\downarrow wav) \setminus (\downarrow w)\big) \cdot a$ sont disjoints si et seulement si le mot v ne contient pas la lettre a.

---

1. *On a $(\downarrow wav) \subset (\downarrow wav) \cdot a \subset (\downarrow wava)$ d'où $(\downarrow wav) \cup \big((\downarrow wav) \setminus (\downarrow w)\big) \cdot a \subseteq (\downarrow wava)$.*

   Réciproquement, soit $u \in \downarrow wava \setminus \downarrow wav$. Il existe donc un plongement de u dans $wava$ mais il n'y a pas de plongement de u dans wav.
   *Si la dernière lettre de u n'était pas a alors, d'après la question **1**, u serait aussi un plongement dans wav, ce qu'on a exclu. En écrivant $u = u'a$, il y a un plongement de u' dans wav mais il n'y a pas de plongement de u dans wa (sinon on en déduirait un plongement de u dans wav) et donc il ne peut pas y avoir de plongement de u' dans w : $u' \in (\downarrow wav) \backslash \downarrow w)$ donc $u \in (\downarrow wav \setminus \downarrow w) \cdot a$. On a donc prouvé la seconde inclusion d'où l'égalité.*

2. *Si u appartient à $(\downarrow wav) \cap \big((\downarrow wav) \setminus (\downarrow w)\big) \cdot a$ alors $u \preccurlyeq wav$*

et $u = u'a$ avec $u' \preccurlyeq wav$ et $u' \not\preccurlyeq w$. On en déduit que $u = u'a \not\preccurlyeq wa$.
Dans ce cas, pour le plongement de u dans wav, le a final de u est nécessairement dans v donc v contient la lettre a.

Réciproquement, si v contient la lettre a, alors on a $waa \preccurlyeq wav$: $waa \in (\downarrow wav)$
De plus $wa \preccurlyeq wav$ et $wa \not\preccurlyeq w$ donc $waa \in \big((\downarrow wav) \setminus (\downarrow w)\big) \cdot a$.
L'intersection $(\downarrow wav) \cap \big((\downarrow wav) \setminus (\downarrow w)\big) \cdot a$ n'est donc pas vide car elle contient waa.

En conclusion, l'union est disjointe si et seulement si le mot v contient la lettre a.

Quand l'union est disjointe dans (‡), on peut obtenir le cardinal de $(\downarrow u)$, pour $u = wava$, en fonction des cardinaux de $(\downarrow wav)$ et de $p(\downarrow w)$. Cette approche se généralise au cas d'un mot u quelconque.

Question 7

1. Donnez des équations récursives permettant de calculer le cardinal de $(\downarrow u)$ 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.
2. En se basant sur vos équations, programmez une fonction OCaml `nb_sousmots : string -> int` qui, pour un mot u donné, calcule le cardinal de $(\downarrow u)$ en temps polynomial en $|u|$.
   Justifiez votre analyse de complexité.

---

1. *Si la dernière lettre a d'un mot u apparaît au moins deux fois, on peut le mettre sous la forme $u = wava$ avec v qui ne contient pas la lettre a. Dans ce cas, d'après la question précédente, on a*
   $\mathrm{Card}(\downarrow wava) = \mathrm{Card}(\downarrow wav) + \mathrm{Card}\big[\big((\downarrow wav) \setminus (\downarrow w)\big) \cdot a\big]$. *Or on a $(\downarrow w) \subset (\downarrow wav)$ et la la multiplication par une lettre conserve le cardinal donc* $\mathrm{Card}(\downarrow wava) = 2\mathrm{Card}(\downarrow wav) - \mathrm{Card}(\downarrow w)$.
   Par contre, si a apparaît une seule fois dans u (en dernière position, donc), en écrivant $u = wa$, on prouve qu'on a l'union disjointe $(\downarrow u) = \big((\downarrow w) \sqcup (\downarrow w)\big) \cdot a$ d'où $\mathrm{Card}(\downarrow u) = 2\mathrm{Card}(\downarrow w)$.
2. *On a besoin de chercher l'avant-dernière position de la dernière lettre*

```
let precedent i ch =
  let j = ref (i-1) in
  let a = ch.[i] in
  while !j >= 0 && ch.[!j] <> a do decr j done;
  !j
```

On conserve l'idée de la programmation dynamique.

```
let nb_sousmots u =
  let n = String.length u in
  let m = Array.make (n+1) 0 in
  m.(0) <- 1;
  for i = 1 to n do
  let j = precedent (i-1) u in
    if j = -1
    then m.(i) <- 2 * m.(i-1)
    else m.(i) <- 2 * m.(i-1) - m.(j) done ;
  m.(n)
```

La fonction obtenue a une complexité temporelle en $\mathcal{O}\big(|u|^2\big)$ de fait de la recherche linéaire

du précédent. La complexité spatiale est en $\mathcal{O}(|u|)$.

Un *sur-mot* commun à u et v est un mot w tel que $u \preccurlyeq w$ et $v \preccurlyeq w$. 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 $\text{pcsmc}(\texttt{informatique}, \texttt{difficile}) = \texttt{difnficormatilque}$.

Question 8

1. 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.
2. Généralisez la propriété précédente en donnant des équations qui permettent de caractériser $\text{pcsmc}(ua, vb)$ dans le cas général, y compris quand $a = b$.
3. 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.

---

1. *Si $pcsmc(ua, vb) = wa$ alors vb est un sous-mot de wa donc de w car $b \neq a$.*
   De plus ua est un sous-mot de wa donc u est un sous-mot de w.
   Ainsi w est un sur-mot commun de u et vb.
   Pour tout sur-mot commun, w', de u et vb on a $w'a$ sur-mot de ua et vb.
   On doit alors avoir $|w'a| \geqslant |wa|$ car wa est minimal donc $|w'| \geqslant |w|$.
   Si on a $|w'| = |w|$ alors la multiplication par a conserve l'ordre donc on doit avoir w avant w' pour l'ordre lexicographique car wa est avant $w'a$.
   Ainsi w est le plus petit sur-mot commun : $w = pcsmc(u, vb)$.
2. - *On a facilement $pcsmc(u, \varepsilon) = u$ et $pcsmc(\varepsilon, v) = v$.*
   - *Si $u = u'a$ et $v = v'b$ avec $a \neq b$ et $w = pcsmc(u'a, v'b)$ alors*
     soit w se termine par a, $w = w'a$, et $w' = pcsmc(u', v'b)$ d'après la question précédente
     soit w se termine par b, $w = w'b$, et $w' = pcsmc(u'a, v')$ de la même façon.
     Ainsi w est le plus petit mot entre $pcsmc(u', v'b) \cdot a$ et $pcsmc(u'a, v') \cdot b$.
   - *Si $u = u'a$ et $v = v'ab$ et $w = pcsmc(u'a, v'a)$ alors w se termine par a, $w = w'a$ et on prouve de même que $w' = pcsmc(u', v')$.*
3. *Toujours la programmation dynamique*

```
let pcsmc u v =
  let n = String.length u in
  let m = String.length v in
  let p = Array.make_matrix (n+1) (m+1) "" in
  for i = 1 to n do p.(i).(0) <- String.sub u 0 i done;
  for j = 1 to m do p.(0).(j) <- String.sub v 0 j done;
  for i = 1 to String.length u do
    for j = 1 to String.length v do
      if u.[i-1] = v.[j-1]
      then p.(i).(j) <- p.(i-1).(j-1)^(Char.escaped u.[i-1])
      else let w1 = p.(i).(j-1)^(Char.escaped v.[j-1]) in
           let w2 = p.(i-1).(j)^(Char.escaped u.[i-1]) in
          if String.length w1 < String.length w2
          then p.(i).(j) <- w1
          else if String.length w2 < String.length w1
               then p.(i).(j) <- w2
               else if w1 < w2
                    then p.(i).(j) <- w1
                    else p.(i).(j) <- w2 done done;
  p.(n).(m)
```

II Sous-mots et expressions rationnelles

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 $((\emptyset|\epsilon)*)*$ 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 chaque langage $L(e_1)$ et $L(e_2)$, dites s'il contient un mot commençant par a ; par b ; par c.

La première expression rationnelle correspond à $((\mathtt{a}|\emptyset)\cdot\varepsilon)*\cdot(\mathtt{b}\cdot(\emptyset\cdot(\mathtt{cc})*))$.
Cette expression contient un facteur $\emptyset$ *donc le langage dénoté est vide :* $L(e_1)$ *ne contient aucun mot commençant par* a*,* b *ou* c*.*
La deuxième correspond à $(\mathtt{ba})*(\epsilon|\mathtt{a})\mathtt{c}^*$*.*
Ainsi, $L(e_2)$ *contient les mot* $ba\cdot\varepsilon\cdot\varepsilon = ba$*,* $\varepsilon\cdot a\cdot\varepsilon = a$ *et* $\varepsilon\cdot\varepsilon\cdot c = c$*.*

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.

Dans le cas d'un produit il faut tenir compte d'un ensemble qui peut être vide et aussi de l'appartenance de ε *au premier langage.*

```
{ocaml}
let rec langage_vide = function
  | Epsilon -> false
  | Empty -> true
  | Letter a -> false
  | Sum(e1, e2) -> langage_vide e1 && langage_vide e2
  | Product(e1, e2) -> langage_vide e1 || langage_vide e2
```

```
let rec contient_epsilon = function
  | Epsilon -> true
  | Empty -> false
  | Letter a -> false
  | Sum(e1, e2) -> contient_epsilon e1 || contient_epsilon e2
  | Product(e1, e2) -> contient_epsilon e1 && contient_epsilon
     e2
  | Star(e1) -> true
```

```
let rec peut_debuter_par e a =
  match e with
  | Epsilon -> false
  | Empty -> false
  | Letter ch -> ch = a
  | Sum(e1, e2)
      ->   peut_debuter_par e1 a || peut_debuter_par e2 a
  | Product(e1, e2)
     ->   (peut_debuter_par e1 a && not (langage_vide e2))
       || (contient_epsilon e1 && peut_debuter_par e2 a)
  | Star(e1) -> peut_debuter_par e1 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 rationelle e, si u est dans $\downarrow L(e)$, c'est-à-dire si u est un 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

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

Ainsi, u est sous-mot d'un mot de L si et seulement si $\epsilon \in \langle u \rangle L$.
Notons d'ailleurs que $\epsilon \in \langle u \rangle L$ 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 langage L, L_1, L_2. Justifiez vos réponses négatives par un contre-exemple.

1. $\langle \epsilon \rangle L = L$,
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. $\langle u \rangle (L^*) = (\langle u \rangle L) \cdot L^*$.

---

1. *Pour $L = \{aa\}$ et $v = w = a$, on a $\epsilon \preccurlyeq w$ et $wv \in L$, donc $a \in \langle \epsilon \rangle L$ mais $a \notin L$.*
2. *Pour $L_1 = \{a\}$ et $L_2 = \{bb\}$, on a $L_1 \cdot L_2 = \{abb\}$.*
   Or $a \preccurlyeq ab = w$ et $wv \in L_1 \cdot L_2$ pour $v = b$ donc $b \in \langle a \rangle (L_1 \cdot L_2)$.
   De plus $\langle a \rangle (L_2) = \emptyset$ car L_2 ne contient aucun mot contenant un a et un mot de $(\langle a \rangle L_1) \cdot L_2$ doit se terminer par bb, donc $b \notin (\langle a \rangle L_1) \cdot L_2$. Ainsi $b \notin (\langle a \rangle L_1) \cdot L_2 \cup \langle a \rangle L_2$.
3. *Pour $L = \{abc\}$, on a $\langle ab \rangle L = \langle b \rangle L = \{\varepsilon, c\}$ mais $\langle a \rangle \{\varepsilon, c\} = \emptyset$.*
   Ainsi, pour $u = a$ et $v = b$, $\langle uv \rangle L \neq \langle u \rangle (\langle v \rangle L)$
4. *$L = \{a\}$, $\langle aa \rangle L = \emptyset$ donc $(\langle aa \rangle L) \cdot L^* = \emptyset$.*
   Cependant $L^* = \{a^n \ ; \ n \in \mathbb{N}\}$ donc $\langle aa \rangle (L^*) = L^* \neq (\langle aa \rangle L) \cdot L^*$.

Ainsi, aucune de ces égalités n'est vraie.

Question 12

1. Programmez une fonction OCaml `eps_residu_ratexp : ratexp -> ratexp` qui, à partir d'une expression rationnelle e construit une expression rationnelle e' telle que $L(e') = \langle\epsilon\rangle L(e)$.
2. Donnez (et justifiez) un majorant, en fonction de $|e|$, de la taille $|e'|$ de l'expression construite par votre programme.

---

1. *$\langle\epsilon\rangle L$ est l'ensemble des suffixes des mots de L.*

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

2.

- *Si* `e` $=$ `Empty` *ou* `e` $=$ `Epsilon` *alors* $|e| = |e'| = 1$.
- *Si* `e` $=$ `Lettera` *alors* $|e'| = |e| + 2 = 3$.
- *Si* `e` $=$ `Sum(`$\mathtt{e}_1, \mathtt{e}_2$`)` *alors* $|e| = |e_1| + |e_2| + 1$ *et* $|e'| = |e'_1| + |e'_2| + 1$.
- *Si* `e` $=$ `Product(`$\mathtt{e}_1, \mathtt{e}_2$`)` *alors* $|e| = |e_1| + |e_2| + 1$ *et* $|e'| \leqslant |e'_1| + |e_2| + |e'_2| + 2$.
- *Si* `e` $=$ `Stare`$_1$ *alors* $|e| = |e_1| + 1$ *et* $|e'| = |e_1| + |e'_1| + 1$.

On cherche $f(n)$ telle que $|e'| \leqslant f(n)$ pour $|e| \leqslant n$.

- *Si* `e` $=$ `Empty` *ou* `e` $=$ `Epsilon` *alors* $|e| = |e'| = 1$ *donc* $f(1) \geqslant 1$.
- *Si* `e` $=$ `Lettera` *alors* $|e'| = |e| + 1 = 2$ *donc* $f(1) \geqslant 3$.
- *Si* `e` $=$ `Sum(`$\mathtt{e}_1, \mathtt{e}_2$`)` *alors* $|e| = |e_1| + |e_2| + 1$ *et* $|e'| = |e'_1| + |e'_2| + 1$.
  Si on note $|e_1| = p$ *et* $|e_2| = q$ *alors* $p + q = n$ *pour* $|e| = n + 1$ *d'où* $f(n+1) \geqslant f(p) + f(q) + 1$
- *Si* `e` $=$ `Product(`$\mathtt{e}_1, \mathtt{e}_2$`)` *alors* $|e| = |e_1| + |e_2| + 1$ *et* $|e'| \leqslant |e'_1| + |e_2| + |e'_2| + 2$.
  Avec les mêmes notations $f(n+1) \geqslant f(p) + f(q) + 1 + q + 1$
- *Si* `e` $=$ `Stare`$_1$ *alors* $|e| = |e_1| + 1$ *et* $|e'| = |e_1| + |e'_1| + 1$.
  On a donc $f(n+1) \geqslant f(n) + n + 1$

En posant $f(0) = 0$ *on veut* $f(n+1) \geqslant n + 1 + \sup\{f(p) + f(q)\ ;\ 0 \leqslant p, 0 \leqslant q, p + q = n\}$.

$f(n+1) \geqslant f(n) + n + 1$ *suggère une expression de degré 2 et* $f(1) \geqslant 3$ *suggère* $f(n) = 3n^2$.
On peut alors prouver que cela convient par récurrence, c'est vrai pour $n = 1$.
Si $f(p) = 3p^2$ *convient pour* $0 \leqslant p \leqslant n$ *alors*
$\sup\{f(p) + f(q)\ ;\ 0 \leqslant p, 0 \leqslant q, p + q = n\} = \sup\{3p^2 + 3(n-p)^2\ ;\ 0 \leqslant p \leqslant n\} = 3n^2$
et on a bien $3(n+1)^2 \geqslant 3n^2 + n + 1$.
On a donc $|e'| \leqslant 3|e|^2$.
Amélioration *Si on suit la trace de* $f(n+1) \geqslant f(n) + n + 1$, *on arrive en fait à* $f(n) = \frac{n^2+n+4}{2}$ *qui convient et donne un meilleur majorant.*

Question 13

1. 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 $L(e'') = \langle a \rangle L(e)$.
2. Donnez (et justifiez) un majorant, en fonction de $|e|$, de la taille $|e''|$ de l'expression construite par votre programme.

---

1. *Dans un produit de 2 langages $L_1 \cdot L_2$, le préfixe w de la définition peut être*
   - *un préfixe d'un mot de L_1, on a donc les mots de $\langle a \rangle L_1 \cdot L_2$,*
   - *$w = w_1 \cdot w_2$ où w_1 est un mot de L_1 contenant a et w_2 est un préfixe d'un mot de L_2, on a donc les mots de $\langle \varepsilon \rangle L_2$,*
   - *$w = w_1 \cdot w_2$ où w_1 est un mot de L_1 et w_2 est un préfixe contenant a d'un mot de L_2, on a donc les mots de $\langle a \rangle L_2$, or on a $\langle a \rangle L_2 \subset \langle \varepsilon \rangle L_2$,*

   Cette analyse fonctionne aussi pour L_1^*.

```
let rec char_residu_ratexp a e =
  match e with
  | Epsilon -> Empty
  | Empty -> Empty
  | Letter ch when ch = a -> Epsilon
  | Letter ch -> Empty
  | Sum(e1, e2) -> Sum(char_residu_ratexp a e1,
     char_residu_ratexp a e2)
  | Product(e1, e2) ->
     if langage_vide e1
     then Empty
     else let e1_prime = char_residu_ratexp a e1 in
       if langage_vide e1_prime
       then char_residu_ratexp a e2
       else Sum(Product(char_residu_ratexp a e1, e2),
                eps_residu_ratexp e2)
  | Star(e1) ->
     if langage_vide (char_residu_ratexp a e1)
     then Empty
     else eps_residu_ratexp e
```

2. *Le même majorant convient, La démonstration doit tenir compte de l'usage de* `eps_residu_ratexp`*.*

Question 14

1. 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`.
2. 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 $0 \leqslant i \leqslant j \leqslant |u|$ et $u[i, j[\in {\downarrow}L$. 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 $M[i, j] =$ `true` ssi $(i, j) \in FC(u, e)$. Notons qu'en particulier

$M[i,j] =$ `false` pour $j < i$.

Question 15 Programmez une fonction
`facteurs_couverts : string -> ratexp -> bool array array` q
ui, 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.