1) a) Procédons par récurrence sur la taille des mots. Étant donné un plongement $p : [m] \to [n]$ de ua dans $u'a'$, deux cas sont possibles :
— si $p(m-1) = n-1$, alors nécessairement

$$a = (ua)[m-1] = (u'a')[n-1] = a'$$

et la restriction $p|_{[m-1]}$ est strictement croissante de $[m-1]$ dans $[n-1]$, i.e. c'est un plongement de u dans u', d'où $u \preccurlyeq u'$;
— sinon, $p\big([m]\big) \subseteq [n-1]$ et donc $ua \preccurlyeq u'$.
Réciproquement, il est clair que, de chaque cas de la disjonction de droite, on peut déduire que $ua \preccurlyeq u'a'$.

b) De la question précédente, on déduit le programme suivant, sachant que l'équivalence que l'on vient de montrer suppose que les deux mots comportent au moins une lettre. Pour raison d'efficacité, la disjonction précédente peut être transformée en deux cas *disjoints* :

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

puisque si $ua \preccurlyeq u'$, alors nécessairement $u \preccurlyeq u'$.

```
let teste_sous_mot v u =
  let rec aux i j =
    if i = 0 then true
    else if j = 0 then false
    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)
;;
```

Concernant la complexité, chaque appel récursif à `aux` se fait en temps constant, et puisque `j` décroît de 1 à chaque fois, le nombre d'appels est majoré par $|u|$. La complexité de la fonction est en $O\big(|u|\big)$.

2) a) On a bien $\binom{\texttt{abab}}{\texttt{ab}} = 3$, puisque c'est le nombre de façons de sélectionner dans `abab` un `a` puis un `b` :

`ab`ab `a`ba`b` `ab`ab

b) La donnée d'un plongement de a^m dans a^n revient à sélectionner m positions dans l'ensemble $[n]$ à n éléments. Il y a $\binom{n}{m}$ façons de faire, d'où :

$$\binom{a^n}{a^m} = \binom{n}{m}.$$

c) C'est une conséquence directe de la question 1a, sachant que les ensembles de plongements de va dans u d'une part, et de v dans u d'autre part sont disjoints (du fait du cardinal de l'ensemble de départ).

3) a) La terminaison de `nb_plongements` est assurée par le fait que les arguments `i` et `j` de la fonction auxiliaire `aux` sont des entiers positifs, et que les appels récursifs à cette même fonction font toujours décroître strictement `j`.

b) La correction de `nb_plongements` découle de la question 2c et, plus généralement, de la question 1a auquel on a ajouté les cas de base :

$$\binom{u}{\varepsilon} = 1 \qquad v \neq \varepsilon \implies \binom{\varepsilon}{v} = 0$$

$$\binom{ua}{va} = \binom{u}{va} + \binom{u}{v} \qquad a \neq a' \implies \binom{ua'}{va} = \binom{u}{va}$$

Ces différentes équations correspondent aux quatre branches de la fonction.

4) a) Soit $(i,j) \in \mathbb{N}^2$. Notons $S(i,j)$ le maximum des valeurs $T(v,u)$ où v décrit les mots de longueur i et u décrit les mots de longueur j.
Si i et j ne sont pas nul on a

$$S(i,j) \leq 1 + S(i-1,j-1) + S(i,j-1)$$

puisque le pire des cas est celui où u et v termine par la même lettre.
On sait de plus que $S(i,0) = 1$ pour toute valeur de i car si $u = \varepsilon$, la fonction `nb_plongements` n'appelle qu'une fois la fonction `aux`. On peut alors conjecturer que pour tout $(i,j) \in \mathbb{N}^2, S(i,j) < 2^j \times 2$ c'est-à-dire $S(i,j) \leq 2^{j+1} - 1$. Montrons cela par récurrence sur l'entier j.
— Si $j = 0$, on a vu que $S(i,0) = 1 \leq 2^1 - 1$ pour tout $i \in \mathbb{N}$.
— Soit $j \in \mathbb{N}$. On suppose que pour tout $i \in \mathbb{N}$, $S(i,j) \leq 2^{j+1} - 1$. En utilisant la formule ci-dessus, pour $i \in \mathbb{N}$,

$$S(i,j+1) \leq 1 + S(i-1,j) + S(i,j) \leq 1 + 2^j - 1 + 2^j - 1 = 2^{j+1} - 1$$

On pose donc $C_1 = 2$. Pour tous mots, u, v, $T(v,u) \leq S(|v|,|u|) < 2^{|u|} \times 2$.

b) 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 tout en ayant $\binom{u}{v} = 0$. On peut considérer par exemple des mots de la forme

$$u = \mathtt{a}^n \qquad v = \mathtt{ba}^m$$

puisque le mot u ne contient pas la lettre `b`, mais l'algorithme, en lisant les lettres de droite à gauche, doit au moins parcourir le mot u en entier.

c) Les cas de bases de la fonction `aux` ne peuvent renvoyer que 0 ou 1. La fonction renvoyant $\binom{u}{v}$ elle a donc été appelée $\binom{u}{v}$ sur des valeurs qui valaient 1. Dit autrement on peut écrire

$$\binom{u}{v} = 1 + 1 + 1 + \ \cdots \ + 1 + 1$$

Dans cette écriture les + correspondent aussi à un appel de la fonction `aux`. ll y en a $\binom{u}{v} - 1$. Cela montre que

$$T(v,u) \geq 2\binom{u}{v} - 1.$$

5) On peut accélérer la fonction en sauvegardant les valeurs obtenus lors des appels récursifs dans une matrice d'entiers. Cela donne le programme suivant :

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

La complexité est alors clairement en $O\big(|u| \times |v|\big)$ en temps comme en espace. En particulier, on a bien un temps polynomial en $|u| + |v|$.

6) a) Remarquons tout d'abord que $\downarrow wav \subseteq (\downarrow wav) \cdot a \subseteq \downarrow wava$, ce qui assure l'inclusion

$$\downarrow wav \cup \big(\downarrow wav \setminus \downarrow w\big) \cdot a \subseteq \downarrow wava$$

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. Cela implique tout d'abord que la dernière lettre de u est a (car sinon, un plongement existe dans wav, toujours d'après la question 1a). En particulier, 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 (on en déduirait un plongement de u dans wav) et donc il n'y a pas de plongement de u' dans w. Ainsi, on a :

$$u \in (\downarrow wav \setminus \downarrow w) \cdot a.$$

On a donc montré que :

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

b) Si $\downarrow wav \cap \big(\downarrow wav \setminus \downarrow w\big) \cdot a \neq \emptyset$, un mot u dans l'intersection se termine par un a (à cause du « $\cdot a$ »). Comme $u \in \downarrow wav$ et, en notant $u = u'a$, $u' \notin \downarrow w$, on a $u \notin \downarrow wa$. Cela entraîne que dans le plongement de u dans wav, le a final de u est nécessairement dans v, et donc que v contient la lettre a.

Réciproquement, si v contient la lettre a, alors on a $waa \in \downarrow wav$ ainsi que $wa \in \downarrow wa \setminus \downarrow w$, d'où

$$waa \in \downarrow wav \cap \big(\downarrow wav \setminus \downarrow w\big) \cdot a$$

En particulier, l'intersection est non vide.

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

7) a) 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

$$\mathrm{Card}(\downarrow wava) = \mathrm{Card}(\downarrow wav) + \mathrm{Card}\big((\downarrow wav \setminus \downarrow w) \cdot a\big)$$

Mais, puisque $\downarrow w \subseteq \downarrow wav$, on a :

$$\mathrm{Card}\big((\downarrow wav \setminus \downarrow w) \cdot a\big) = \mathrm{Card}\big(\downarrow wav \setminus \downarrow w) = \mathrm{Card}(\downarrow wav) - \mathrm{Card}(\downarrow w)$$

Ainsi,

$$\mathrm{Card}(\downarrow wava) = 2\mathrm{Card}(\downarrow wav) - \mathrm{Card}(\downarrow w)$$

Par contre, si a apparaît une seule fois dans u (en dernière position, donc), en écrivant $u = wa$, il est clair que l'on a l'union disjointe

$$\downarrow u = \downarrow w \cup (\downarrow w) \cdot a.$$

On a alors

$$\mathrm{Card}(\downarrow u) = 2\mathrm{Card}(\downarrow w)$$

b) En suivant l'idée de programmation dynamique, on peut calculer efficacement le nombre de sous-mots en utilisant un tableau pour stocker le nombre de sous-mots des différents préfixes de `u` :

```
let nb_sousmots (u : string) =
  let m = Array.make (String.length u + 1) 0 in
  m.(0) <- 1 ;
  for i = 1 to String.length u do
    let a = u.[i - 1]
    and j = ref (i - 2) in
    while !j >= 0 && u.[!j] != a do
      decr j
    done ;
    if !j = -1 then
      (* a apparaît une unique fois *)
      m.(i) <- 2 * m.(i - 1)
    else
      m.(i) <- 2 * m.(i - 1) - m.(!j)
  done ;
  m.(String.length u)
;;
```

La fonction obtenue a une complexité temporelle en $O\big(|u|^2\big)$ de fait des deux boucles imbriquées, et une complexité spatiale en $O\big(|u|\big)$.

Remarque : à l'aide d'un dictionnaire qui indiquerait, pour chaque lettre, sa dernière occurence sous réserve de définition, la fonction précédente pourrait avoir une complexité en $O\big(|u|\big)$.

8) a) Si $\mathrm{pcsmc}(ua, vb) = wa$ alors vb est un sous-mot de w (puisque $b \neq a$) et comme ua est un sous-mot de wa, alors nécessairement u est un sous-mot de w. Mais si $w \neq \mathrm{pcsmc}(u, vb)$, alors $\mathrm{pcsmc}(u, vb)$ est strictement plus petit que w pour l'ordre lexicographique, et il en est de même pour $\mathrm{pcsmc}(u, vb) \cdot a$ par rapport à wa. On a donc nécessairement $w = \mathrm{pcsmc}(u, vb)$.

b) Clairement, on a $\mathrm{pcsmc}(ua, va) = \mathrm{pcsmc}(u, v) \cdot a$ alors que, dans le cas où les deux mots se terminent par des lettres différentes, on a soit $\mathrm{pcsmc}(ua, vb) = \mathrm{pcsmc}(u, vb) \cdot a$ ou bien $\mathrm{pcsmc}(ua, vb) = \mathrm{pcsmc}(ua, v) \cdot b$ (d'après la question précédente, sachant que la dernière lettre de $\mathrm{pcsmc}(ua, vb)$ est nécessairement a ou b pour ne pas avoir de lettre inutile). Pour déterminer quelle possibilité suivre, il suffit de comparer les deux chaînes, et d'en prendre le plus petit selon la relation d'ordre indiquée : taille puis ordre lexicographique.

c) On en déduit la fonction suivante, toujours en utilisant les idées de programmation dynamique :

```
let pcsmc (u : string) (v : string) =
  let m = Array.make_matrix (String.length u + 1)
                            (String.length v + 1) "" in
  for i = 1 to String.length u do
    m.(i).(0) <- String.sub u 0 i
  done ;
  for j = 1 to String.length v do
    m.(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
        m.(i).(j) <- m.(i - 1).(j - 1) ^ (String.sub u (i - 1) 1)
      else
        let w1 = m.(i).(j - 1) ^ (String.sub v (j - 1) 1)
        and w2 = m.(i - 1).(j) ^ (String.sub u (i - 1) 1)
        in
        if String.length w1 < String.length w2 then
          m.(i).(j) <- w1
        else if String.length w2 < String.length w1 then
          m.(i).(j) <- w2
        else (* w1 et w2 sont de même taille *)
        if w1 < w2 then m.(i).(j) <- w1 else m.(i).(j) <- w2
    done
  done ;
  m.(String.length u).(String.length v)
;;
```

À nouveau, les complexités temporelle et spatiale sont en $O\big(|u| \times |v|\big)$, en supposant que que les opérations sur les chaînes de caractères sont en temps constant. De façon plus réaliste, si l'on suppose que ces opérations sont de complexité linéaire en la taille des chaînes de caractères, la complexité de chacune de ces opérations est en $O\big(|u| + |v|\big)$. On obtient encore une complexité polynomiale en $|u| + |v|$.