Épreuve d'informatique – Concours ENS 2003

Guillaume Munch

Version de travail : 26 novembre 2006

Corrigé

Nous avons choisi d'écrire les algorithmes dans un pseudo-Caml en conservant les notations de l'énoncé. En particulier, les indices commencent à 1.

Correction de l'énoncé

– À la question 2.3.c., il faut lire :

$$j = \min\left(\{|\mathtt{s}|+1\} \cup \{l > d_{k-1} \mid s[l] \neq s[l-k+1]\}\right)$$

– Dans l'introduction de la partie 3, nous rajoutons la contrainte suivante dans la définition d'un chemin : si un chemin s'arrête en position z du nœud ν qui n'est pas la racine, alors $z \geq debut(\nu)$.
Sans cette contrainte, l'étiquette depuis la racine d'un nœud ν qui n'est pas la racine pourrait être obtenu par deux chemins différents : l'un s'arrêtant au père de ν, l'autre s'arrêtant en position $debut(\nu) - 1$ du nœud ν, or ce n'est pas ce que l'on souhaite.

Partie 1

1.1.

```
let compare_sous_chaine s t i j k =
  if i=0 || j=0 || i+k>|s| || j+k>|t| then
    false
  else (
    let k' = ref 0
    in
    while s[i+k'] = t[j+k'] && k' <= k do
      incr k'
    done ;
    k' - 1 = k
  )
;;
```

$k' - 1$ est le rang du précédent caractère identique dans les deux sous-chaînes. Donc $\forall k' \in [0, k], s[i + k'] = t[j + k']$ si et seulement si $k' - 1 = k$.

La complexité en est linéaire : le pire des cas coûte $k + 1$ comparaisons ; il est atteint si $\forall k' \in [0, k - 1], s[i + k'] = t[j + k']$.

1.2. Soit t un mot tel que $\mathcal{I}_t \geq 1$. Il existe $i \in \mathcal{I}_t$. Alors, par définition, t est de la forme ucv avec $u \in \mathcal{A}^{i-1}$ et $v \in \mathcal{A}^{n-m-i+1}$, donc $t \in \mathcal{A}^* c \mathcal{A}^*$.

Réciproquement, un mot de $\mathcal{A}^* c \mathcal{A}^*$ contient une occurence de t. Donc l'ensemble des mots t tels que $\operatorname{card} \mathcal{I}_t \geq 1$ est $\mathcal{A}^* c \mathcal{A}^*$, qui est bien rationnel.

$\overline{\mathcal{A}^* c \mathcal{A}^*}$ est l'ensemble des mots t tels que $\operatorname{card} \mathcal{I}_t = 0$. Il est rationnel puisque le complémentaire d'un langage rationnel est rationnel.

1.3. Ci-dessous, on empile les occurrences dans `result`. On inverse la liste obtenue pour obtenir les résultats dans l'ordre croissant.

```
let occurences c t =
  let result = ref []
  in
  for i=1 to n do
    if compare_sous_chaine c t 1 i (|c|-1) then
      result := i :: !result
  done ;
  list_rev !result
;;
```

`compare_sous_chaine` est en $O(m)$ et est évalué n fois, d'où un coût de $O(nm)$.

1.4. On remarquera que la définition de sous chaîne commune est erronée : le mot u y est de longueur $l+1$!

Notre algorithme cherche pour tout couple d'indices i, j des chaînes s et t s'il existe une sous-chaîne commune de longueur plus grande que l_{max} qui est le « l » (cf. définition) de la plus grande sous-chaîne commune trouvée jusque-là.

```
let plus_longue_sous_chaine_commune s t =
  let l_max = ref (-1)
  in
  let debut = ref 0
  in
  for i=1 to |s| do
    for j=1 to |t| do
      for k = !l_max + 1 to |t| - j do
        if compare_sous_chaine s t i j k then (
          max := k ;
          debut := i
        )
      done
    done
  done ;
  s[ !debut .. !debut + !l]
;;
```

Par défaut, ε est la plus longue sous chaîne commune ; dans ce cas, `l_max` vaut -1 et l'algorithme renvoie bien ε.

Justification du coût Chaque itération de la boucle sur k est $O(|\mathtt{t}|)$ car $k \leq |\mathtt{t}|$ et que l'opération de poids dominant est l'appel à `compare_sous_chaine`. De plus, la boucle sur k effectue moins de $|\mathtt{t}|$ itérations. D'où un coût en $O(|\mathtt{s}|\,|\mathtt{t}|^3)$ pour les trois boucles imbriquées. C'est le coût de l'algorithme.

1.5. Pour i_1 et $i_2 > i_1$ donnés, on regarde s'il y existe un triplet de répétition maximale (i_1, i_2, k) où k croît de 0 tant que c'est le cas. Les deux premières boucles assurent que

ceci est effectué pour tous i_1 et $i_2 > i_1$ en tant que `i` et `j`. Si $s[i_1 - 1] = s[i_2 - 1]$, cela signifie que la répétition s'étend vers la droite, donc qu'elle a déjà été trouvée.

```
let repetitions_max s =
  let result = ref []
  in
  for i = 1 to |s| do
    for j = i+1 to |s| do
      if i=1 || s[i-1] <> s[j-1] then (
        let k = ref 0
        in
        while j + !k <= n && s[i + !k] = s[j + !k] do
          incr k
        done ;
        (* à la sortie de la boucle, on a s[i+k]<>s[j+k] *)
        if !k <> 0 then result := s[i .. i + !k - 1] :: !result
      )
    done ;
  done ;
 !result
;;
```

Sa complexité est en $O(|\mathtt{s}|^3)$ car l'itération sur la troisième boucle imbriquée se fait en temps constant.

Pour retirer les doublons, il suffit d'effectuer un tri par insertion par exemple, qui se charge de les éliminer. Celui-ci est convenable car de coût $o(n^3)$.

L'algorithme est implémenté en OCaml dans le fichier `ml` joint.

Partie 2

2.1. Pour le mot *aabcaabdaaabc* :

i	2	3	4	5	6	7	8	9	10	11	12	13
$L(i)$	1	0	0	3	1	0	0	2	4	1	0	0
g_i	2	2	2	5	5	5	5	9	10	10	10	10
d_i	2	2	2	7	7	7	7	10	13	13	13	13

2.2.

Si $i = 1$ **:** On a $s[k] \neq s[1]$ donc $L(k) = 0$. Donc :

$$d_k = \max_{\substack{1 < j \leq k-1 \\ L(j) \neq 0}} (j + L(j) - 1) = d_{k-1}$$

et $g_k = g_{k-1}$ lorsque le max est défini, et $d_k = 0 = d_{k-1}$ et $g_k = 0 = g_{k-1}$ par définition sinon.

Si $i > 1$ **:** On a par définition de i, $s[k..k+i-2] = s[1..i-1]$ et $s[k+i-1] \neq s[i]$ donc $L(k) = i - 1$. Or $k > d_{k-1}$ signifie pour tout $j < k$, $j + L(j) - 1 < k$ donc aussi $j + L(j) < k + \underbrace{L(k)}_{\geq 1}$. Donc :

$$\begin{aligned} d_k &= k + L(k) - 1 = k + i - 2 \\ g_k &= k \end{aligned}$$

car k est le seul indice à atteindre le maximum dans d_k.

2.3.a. On a $s[g_{k-1}..d_{k-1}]$ préfixe de s, autrement dit :

$$s[g_{k-1}..d_{k-1}] = s[1..L(g_{k-1})]$$

Or $g_{k-1} \leq k - 1$ par définition, on peut donc omettre les $k - g_{k-1}$ premières lettres pour obtenir :

$$s[k..d_{k-1}] = s[k - g_{k-1} + 1..L(g_{k-1})]$$

2.3.b. On a par définition de L :

$$\begin{aligned} s[1..L(k - g_{k-1} + 1)] &= s[k - g_{k-1} + 1..L(k - g_{k-1} + 1) + k - g_{k-1}] \qquad (1) \\ s[L(k - g_{k-1} + 1) + 1] &\neq s[L(k - g_{k-1} + 1) + k - g_{k-1} + 1] \end{aligned}$$

Si l'on suppose $L(k - g_{k-1} + 1) < d_{k-1} - k + 1$, on a $L(k - g_{k-1} + 1) + k - g_{k-1} < d_{k-1} - g_{k-1} + 1 = L(g_{k-1})$. Donc d'après a., on a en particulier :

$$s[k..\underbrace{d_{k-1} - L(g_{k-1}) + L(k - g_{k-1} + 1) + k - g_{k-1}}_{=L(k-g_{k-1}+1)-1+k}] = s[k - g_{k-1} + 1..\underbrace{L(k - g_{k-1}) + k - g_{k-1}}_{<L(g_{k-1})}]$$

et

$$s[L(k-g_{k-1}+1)+k] \quad = \quad s[L(k-g_{k-1}+1)+k-g_{k-1}+1]$$

D'où d'après (1) par transitivité :

$$\begin{aligned} s[1..L(k-g_{k-1}+1)] &= s[k..L(k-g_{k-1}+1)+k-1] \\ s[L(k-g_{k-1}+1)+1] &\neq s[L(k-g_{k-1}+1)+k] \end{aligned}$$

Donc $L(k) = L(k-g_{k-1}+1)$ par définition de $L(k)$. On a donc aussi $L(k) < d_{k-1}-k+1$ ou encore $d_{k-1} > k + L(k) - 1$, donc $d_k = d_{k-1}$ et aussi k n'atteint pas le max dans la définition de d_k, donc $g_k = g_{k-1}$.

2.3.c. Supposons $L(k-g_{k-1}+1) \geq d_{k-1}-k+1$. On a alors $L(k-g_{k-1}+1)+k-g_{k-1} \geq d_{k-1} - g_{k-1} + 1 = L(g_{k-1})$. Donc de manière similaire au b., on obtient :

$$s[k..d_{k-1}] = s[1..d_{k-1}-k+1]$$

Donc $L(k) \geq d_{k-1} - k + 1$. D'où k atteint le max dans la définition de d_k et on a $d_k = k + L(k) - 1$, $g_k = k$.

Rappel : On a défini $j = \min\left(\{|\mathsf{s}|+1\} \cup \{l > d_{k-1} \,|\, s[l] \neq s[l-k+1]\}\right)$.

On a par définition $L(k) = \min\left(\{|\mathsf{s}|+1\} \cup \{l \geq k \,|\, s[l] \neq s[l-k+1]\}\right) - k$. Donc $d_k = k + L(k) - 1 = \min\left(\{|\mathsf{s}|+1\} \cup \{k \leq l \leq |\mathsf{s}| \,\,|\, s[l] \neq s[l-k+1]\}\right) - 1$. Or $d_{k-1} \leq d_k$ donc :

$$\begin{aligned} d_k &= \min\left(\{|\mathsf{s}|+1\} \cup \{k \leq l \leq |\mathsf{s}| \,\,|\, s[l] \neq s[l-k+1]\}\right) - 1 \\ &= \min\Big(\{|\mathsf{s}|+1\} \cup \{\underline{d_{k-1}+1} \leq l \leq |\mathsf{s}| \,\,|\, s[l] \neq s[l-k+1]\}\Big) - 1 \\ &= j - 1 \end{aligned}$$

Donc aussi $L(k) = j - k$, ce que l'on cherchait à exprimer.

2.4. On suppose que le premier indice d'un vecteur est 1.

```
let lgd s =
  let l,g,d = make_vect |s| 0, make_vect |s| 0, make_vect |s| 0
  in
```

```
(* boucle pour k >= 2. Si k = 2 on est dans le
 * premier cas car on a par convention ?.(1) = 0
 *)
for k = 2 to |s| do
  if k > d.(k-1) then (
    let i = ref 1
    in
    while k + !i - 1 <= |s| && s.[k + !i - 1] = s.[!i]
    do
      incr i
    done ;
    if !i = 1 then (
      l.(k) <- 0 ;
      d.(k) <- d.(k-1) ;
      g.(k) <- g.(k-1)
    ) else (
      l.(k) <- !i - 1 ;
      d.(k) <- k + !i - 2 ;
      g.(k) <- k
    )
  ) else (
    (* k <= d.(k-1). On a nécéssairement k >= 3 ici. *)
    if l.(k - g.(k-1) + 1) < d.(k-1) - k + 1 then (
      (* d.(k-1) <> 0 donc g.(k-1) >= 2 *)
      l.(k) <- l.(k - g.(k-1) + 1) ;
      d.(k) <- d.(k-1) ;
      g.(k) <- g.(k-1)
    ) else (
      let j = ref (d.(k-1) + 1)
      in
      while !j <= |s| && s.[!j] = s.[!j - k + 1] do
        incr j
      done ;
      l.(k) <- !j - k ;
      d.(k) <- !j - 1 ;
```

```
        g.(k) <- k
      )
    )
  done ;
  l,g,d
;;
```

L'algorithme est implémenté en OCaml dans le fichier `ml` *joint.*

Justification du coût L'algorithme présente deux invariants :

- Il n'y a de comparaison de caractères que si l'indice j le plus grand des deux caractères comparés vérifie $j > d_{k-1}$, à la k-ème itération.
- Si une comparaison, à la k-ème itération, avec un caractère d'indice $j > d_{k-1}$ est réussie (*true*), alors $d_k \geq j$.

Ceci montre que la suite des indices les plus grands lors des comparaisons de caractères réussies est strictement croissante avec k : il y a donc au plus $|\mathtt{s}|$ comparaisons réussies.

De plus, chaque itération cause au plus une comparaison échouant, donc $|\mathtt{s}|$ au maximum. D'où $O(|\mathtt{s}|)$ comparaisons de caractères.

2.5. Il suffit de calculer L pour le mot $P\#T$ et d'établir la liste des i tels que $L.(i + |\mathtt{P}| + 1) = |\mathtt{T}|$. En effet, on a $i \in \mathcal{I}_t \Leftrightarrow L.(i + |\mathtt{P}| + 1) = |\mathtt{P}|$.

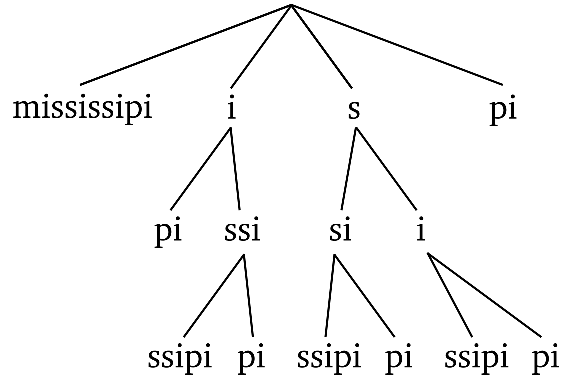
Démonstration. ($\Rightarrow$) est clair par définition de $\mathcal{I}_t$. ($\Leftarrow$) : Si $i \in \mathcal{I}_t$, alors $L(i+|\mathtt{P}|+1) \geq |\mathtt{P}|$. Mais $\# = s[|\mathtt{P}| + 1] \neq s[i + 2\,|\mathtt{P}| + 2] \in \mathcal{A}$. Donc $L(i + |\mathtt{P}| + 1) = |\mathtt{P}|$. □

L'algorithme correspondant a un coût en $O(|\mathtt{P}| + |\mathtt{T}|)$: $O(|\mathtt{P}| + |\mathtt{T}|)$ pour la recherche dans $P\#T$, $O(|\mathtt{T}|)$ pour établir la liste.

L'algorithme est implémenté en OCaml dans le fichier `ml` *joint.*

Partie 3

3.1. Voici l'arbre que l'on obtient :



3.2. Pour commencer, l'invariant suivant est vérifié : *un appel quelconque à $ajoute$ ne modifie pas les chemins existant dans l'arbre.* En effet, dans le cas $flag = true$ et $z \neq fin(B)$, tout chemin $s[a..b]$ se terminant en position z' du nœud B continue d'exister, en particulier dans le cas $z' > z$ où il se terminera en position z'du nouveau fils $(z+1, fin(B), fils(B))$. Et de même, tous les chemins $s[a..b]$ se terminant au-delà de B restent des chemins se terminant au-delà de B, car les fils de B ont la même étiquette depuis la racine avant et après qu'ils ne soient déplacés comme fils du nouveau nœud $(z+1, fin(B), fils(B))$.

On montre le résultat « *pour $i \geq 2$, $1 \leq j \leq i$, avant l'appel à $ajoute(A, s, i, j)$ il existe dans A un chemin d'étiquette $s[j..i-1]$* » par récurrence sur i pour $s \in \mathcal{A}^*$ donné.

Pour $i = 2$: L'arbre est initialisé à :

1,0

|

1,|s|

donc il existe un chemin $s[1] = s[1..i-1]$ et un chemin $\varepsilon = s[2..i-1]$ par définition. D'où le résultat avant l'appel à $ajoute$, un appel quelconque à $ajoute$ ne modifiant pas les chemins existant.

Soit $i \in [3, |\mathsf{s}|]$. Supposons que pour tous i', j' tels que $i > i' \geq 2$ et $1 \leq j \leq i'$, avant l'appel à $ajoute(A, s, i', j)$, il existe dans A un chemin d'étiquette $s[j'..i'-1]$. Soit j tel que $1 \leq j \leq i$.

– Si $j = i$, alors il existe un chemin d'étiquette $\varepsilon = s[j..i-1]$ comme ci-dessus.

- Sinon, $j \leq i-1$, donc avant l'appel à $ajoute(A, s, i-1, j)$, il existe dans A un chemin d'étiquette $s[j..i-2]$.

  Plaçons-nous après l'exécution de $ajoute(A, s, i-1, j)$. On pose $(B, flag, z) = trouve(A, s, i-1, j)$. Si $\neg flag$, alors il existe un chemin d'étiquette $s[j..i-1]$. Sinon, $flag$. Dans ce cas,
  - Si $z = fin(B)$, alors le nouveau fils $(i-1, |\mathsf{s}|, [])$ a une étiquette depuis la racine $s[j..i-1]$.
  - Sinon, si $z \neq fin(B)$, alors B est modifié et ses fils ont des étiquettes depuis la racine $s[j..i-2]$, car le chemin issu de la racine $s[j..i-2]$ se terminait en position z du nœud B avant que celui-ci ne soit modifié. En particulier, il existe un chemin depuis la racine $s[j..i-1]$ qui se termine en position $i-1$ du nouveau nœud $(i-1, |\mathsf{s}|, [])$.

  Dans tous les cas, après l'appel à $ajoute(A, s, i-1, j)$, il existe un chemin d'étiquette $s[j..i-1]$.

Un appel quelconque à $ajoute$ ne modifiant pas les chemins existants, avant l'appel à $ajoute(A, s, i, j)$, il existe un chemin $s[j..i-1]$.D'où le résultat au rang i.

Cqfd par récurrence.

3.3. Soit $s \in \mathcal{A}^+$.

- La propriété « *si* $(u, v, [A_1, \ldots, A_k])$ *est un nœud interne de A, alors pour tout* $1 \leq l < m \leq k$, $s[debut(A_l)] \neq s[debut(A_m)]$ » () est trivialement vraie pour le A initial $(1, 0, [(1, |\mathsf{s}|, [])])$. Soit A vérifiant la propriété (). Soit $i \geq 2$ et j tel que $1 \leq j \leq i$. On pose $(B, flag, z) = trouve(A, s, i, j)$.

  Si $\neg flag$, alors $ajoute(A, s, i, j)$ ne modifie pas A. *Sinon, si* $flag$, cela signifie que, le chemin $s[j..i-1]$ existant, celui-ci se termine en position z du nœud n. Donc :
  - *Si* $z = fin(B)$, alors $\forall A' \in fils(B), s[debut(A')] \neq s[i]$, car sinon il existerait un chemin issu de la racine d'étiquette $s[j..i]$, or $flag \neq false$. Donc « $fils(B) \leftarrow (i, |\mathsf{s}|, []) :: fils(B)$ » conserve la propriété ().
  - *Si* $z \neq fin(B)$, alors on a $s[z+1] \neq s[i]$, car sinon, il existerait un chemin $s[j..i]$. Donc « $fils(B) \leftarrow [(z+1, fin(B), fils(B)), (i, |\mathsf{s}|, [])]$ » conserve la propriété ().

  Dans tous les cas, la propriété () est stable par appel à $ajout(A, s, i, j)$. D'où le résultat par récurrence sur le nombre d'appels à $ajout(A, s, i, j)$: à tout moment, si $(u, v, [A_1, \ldots, A_k])$ est un nœud interne de A, alors pour tout $1 \leq l < m \leq k$, $s[debut(A_l)] \neq s[debut(A_m)]$.
- Soit $t \in A^*$. Soient ν et ν' d'étiquettes dépuis la racine e et e' telles que e et e' sont

des préfixes de t et t préfixe de $e.s[debut(\nu)..fin(\nu)]$. Il existe $\nu_1, \dots, \nu_k$ et $\nu'_1, \dots, \nu'_l$ des nœuds de l'arbre tels que

$$e = s[debut(\nu_1)..fin(\nu_1)].s[debut(\nu_2)..fin(\nu_2)] \cdots s[debut(\nu_k)..fin(\nu_k)]$$

$$e' = s[debut(\nu'_1)..fin(\nu'_1)].s[debut(\nu'_2)..fin(\nu'_2)] \cdots s[debut(\nu'_l)..fin(\nu'_l)]$$

et tels que

$$\forall i \in \{1, \dots, k\}, \nu_{i+1} \text{ est un fils de } \nu_i$$

$$\forall i \in \{1, \dots, l\}, \nu'_{i+1} \text{ est un fils de } \nu'_i$$

où l'on a posé $\nu_{k+1} = \nu$ et $\nu'_{l+1} = \nu'$, et tels que ν_1 et ν'_1 sont la racine.
On pose alors $\mathcal{J} = \{i \mid \nu_i = \nu'_i\}$, non vide car $1 \in \mathcal{J}$, et $j = \max \mathcal{J}$. On a en particulier $j \leq \min(l, k) + 1$.

- Si $j < \min(k, l)+1$, alors on a par définition du chemin $s[debut(\nu_{j+1})] = s[debut(\nu'_{j+1})]$. En particulier dans le cas $j = \min(k, l)$, la contrainte selon laquelle un chemin s'arrêtant en position z du nœud ν vérifie $z \geq debut(\nu)$ assure cette égalité. Donc $\nu_{j+1} = \nu'_{j+1}$, impossible.
- Sinon, $j = \min(k, l) + 1$. Or, la contribution d'un nœud autre que la racine à un chemin n'est jamais nulle. En effet, d'une part, la contrainte selon laquelle un chemin s'arrêtant en position z du nœud ν vérifie $z \geq debut(\nu)$ assure ceci pour les extrémités du chemin. D'autre part, la fonction *ajoute* conserve la propriété $debut(\nu) \leq fin(\nu)$ pour tout nœud ν autre que la racine, qui est vérifiée à l'initialisation, et donc à tout moment.

  Comme les deux chemins ont la même étiquette, donc deux étiquettes de même longueur, ils ont la même profondeur, soit $k = l$.

Ceci montre $\nu = \nu'$, cqfd.

3.4. On a déjà vu que la contribution d'un nœud autre que la racine à un chemin n'est jamais nulle. Donc, à tout moment, comme les étiquettes depuis la racine des nœuds sont des sous-mots de s, il y a au plus $|\mathsf{s}| + 1$ nœuds dans une branche. La complexité de *trouve* est propostionnelle au nombre de nœuds se trouvant sur le chemin, elle est donc linéaire en $|\mathsf{s}|$. Donc celle de *ajoute* également. Or arbre_suffixe évalue $\frac{(|\mathsf{s}|-1)(|\mathsf{s}|+2)}{2}$ fois la fonction *ajoute*.

Donc arbre_suffixe a une complexité en $O(|\mathsf{s}|^3)$.

3.5. Supposons que l'on soit dans le cas 1 au stade l de l'étape m. On pose $\nu = B$ et $k = z$ pour ce stade.

Si pour $j \in \{l+1, \dots, m\}$, on a de nouveau $\nu = B$, alors $z < k$ car le sous-mot est plus court, et donc on est dans le cas $z \neq fin(B)$. Dans ce cas, le nouveau fils $(z+1, fin(B), fils(B))$ vérifie $fils(B) = []$ et on le nomme ν à la place de l'ancien ν.

Donc, qu'il y ait ou non cette modification de ν, puisque $k = m$ et $fils(\nu) = []$ impliquent $fin(\nu) = |\mathtt{s}|$ et $trouve(A, s, m+1, l) = (\nu, false, k+1)$, on est dans le cas 1 au stade l de l'étape $m+1$ si ν n'est pas modifié pour $i = m+1$ et $j \in \{1, \dots, l-1\}$. Ceci est le cas, car si pour un tel j on a $B = \nu$ et $flag = true$, alors cela signifie qye $s[j..i-1] = e.s[debut(\nu)..z]$, donc $s[l..z] = s[j..i-1]$ et donc, comme $j < l-1$, on a $z > i$. Ceci est impossible car on a toujours $z \leq i$: les suffixes sont insérés dans l'arbre avec i croissant.

Donc on est dans le cas 1 au stade $l$de l'étape $m+1$. D'où le résultat par récurrence.

3.6. Supposons que l'on soit dans le cas 3 au stade l de l'étape m. On pose $v = B$, $k = z$. On a donc $fils(\nu) \neq []$ ou $k < m$.

- *Si* $k < m$, alors cela signifie $s[l..m] = s[l-m+k..k]$ avec des indices différents. Alors, les stades $l-m+k+1$ à k de l'étape k ont déjà ajouté les sous-mots

$$s[l-m+k+a..k] = s[l+a..m]$$

  donc on se retrouve dans le cas 3 aux stades $l+a$ de l'étape m.
- *Si* $fils(\nu) \neq []$, alors de manière similaire, le sous-mot qui a ajouté un fils à ν possède des suffixes qui auront ajouté des fils aux B des stades $l+a$ de l'étape m. On s'y retrouve donc également au cas 3.

Cqfd.

3.7. On modifie la fonction ajoute de la manière suivante : elle renvoie 1, 2 ou 3 suivant le cas dans lequel on s'y trouve. On modifie également la fonction arbre_suffixe de la manière suivante : On initialise à *false* deux tableaux cas_1 et cas_3 de longueur $|\mathtt{s}|$. A chaque fois que la fonction ajoute renvoie 1 (resp. 3), on met *true* dans la j-ème (resp i-ème) case du tableau correspondant. Et par la suite on n'appelle la fonction ajoute que si ces cases sont les deux à *false*. En effet, d'après les questions précédentes, on sait que l'on se trouve l'un des deux cas si l'une des cases est à *true,* et dans ce cas on sait que la fonction ajoute ne modifie pas l'arbre.

L'algorithme est implémenté en OCaml dans le fichier `ml` *joint.*

La modification apportée à ajoute ne change pas son coût. Mais maintenant, la double-boucle dans arbre_suffixe effecture $O(|\mathsf{s}|)$ appels à ajoute. En effet, soit dans ajoute flag vaut vrai et un nouveau suffixe est ajouté dans l'arbre, éventualité qui se produit donc $|\mathsf{s}| - 1$ fois au maximum, soit ajoute retourne 1 ou 3, et dans ce cas, une case de cas_1 ou de cas_3 voit sa valeur changée de false à true, ce qui arrive au plus $2\,|\mathsf{s}|$ fois.

Donc arbre_suffixe a une complexité en $O(|\mathsf{s}|^2)$.

3.8 Au début du stade $j+1$, il existe un chemin d'étiquette $s[j+1..i]$. Donc :

- S'il existe un chemin d'étiquette $s[j+1..i+1]$:
  Comme il n'existe pas de chemin d'étiquette $s[j..i+1]$ au début du stade j (cas 2), cela signifie qu'il existe i', j' avec $j' < j$ et $i' < i$ tel que $s[j'+1..i'+1] = s[j+1..i+1]$ de manière à ce qu'à la fin du stade $j'+1$ de l'étape $i'+1$ il existe un chemin d'étiquette $s[j+1..i+1]$.
  Or, comme il existait un chemin d'étiquette $s[j..i]$ au début du stade j, il existe $j'' < j$ et $i'' < i$ tel que $s[j''..i''] = s[j..i]$ et $s[i''+1] \neq s[i+1]$ (cas 2).
  Donc il existe à la fin du stade $j''+1$ de l'étape $i''+1$ un chemin d'étiquette $s[j+1..i].s[i''+1]$. Par unicité du chemin d'étiquette $s[j+1..i]$, il existe ν_1, ν_2 tels que les chemins d'étiquette $s[j+1..i+1]$ et $s[j+1..i].s[i''+1]$ se terminent respectivement en positions $debut(\nu_1)$ et $debut(\nu_2)$ des nœuds ν_1 et ν_2.
  ν_1 et ν_2 sont deux nœuds d'étiquette depuis la racine $s[j+1..i]$.
- Sinon, s'il n'existe pas de chemin d'étiquette $s[j+1..i+1]$: Alors on est dans le cas 2 et ajoute(A,s,i+1,j+1) aboutit à la création d'un nœud d'étiquette depuis la racine $s[j+1..i]$.

D'où le résultat.

3.9. Soit ν_1 un nœud disposant d'un lien suffixe vers ν_2. Il existe i, j tels que l'étiquette depuis la racine de ν_1 soit $s[j..i]$. On pose :

$$\mathcal{N} = \{\nu \,|\, \nu \text{ est sur le chemin menant à } \nu_1\} \setminus fils(A)$$

(l'ensemble des nœuds menant à ν_1 sauf le premier, y compris ν_1). On a card$\,\mathcal{N} = prof(\nu_1) - 1$.

Soit $\nu \in \mathcal{N}$. ν a un chemin d'étiquette depuis la racine $s[j..k]$.

- Si $\nu = \nu_1$, on pose $\varphi(\nu) = \nu_2$.
- Sinon, puisque $prof(\nu) \geq 2$, il a été crée lors d'un stade j' d'une étape k', donc tels que $s[j'..k'] = s[j..k]$. Donc au plus tard à la fin du stade $j'+1$ de l'étape k' il existe

un nœud $\varphi(\nu)$ de chemin depuis la racine $s[j+1..k]$.

On a ainsi défini une injection de $\mathcal{N}$ dans $\{\nu \mid \nu$ est sur le chemin menant à $\nu_2\}$; injection car les $\varphi(\nu)$ ont des chemins depuis la racine distincts.

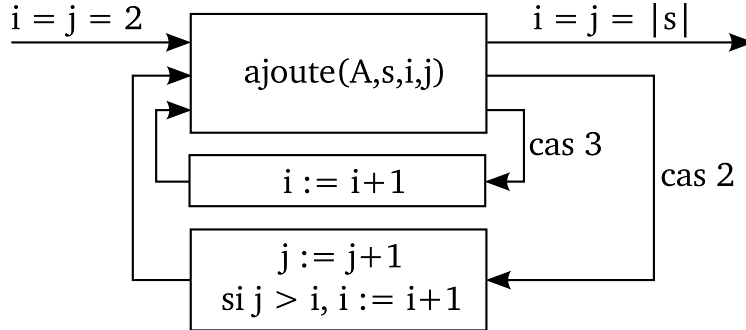
D'où $prof(\nu_1) - 1 \leq prof(\nu_2)$, ce qu'il fallait démontrer.

3.10. On montre que la j-ème création de nœud dans ajoute (c'est-à-dire un cas 2) a lieu lors d'un stade $j+1$.

- « $A \leftarrow (1, 0, [(1, |\mathtt{s}|, [])])$ » peut être vu comme la 0-ème création de nœud puisqu'elle se fait suivant la même procédure que dans ajoute, et on peut considérer qu'elle a lieu au stade 1 de l'« étape 1 ». Mais l'hérédité de la propriété est vraie à l'étape 1 pour les mêmes raisons qu'à l'intérieur de la boucle.
- Supposons que pour $i \in [1, |\mathtt{s}|]$, pour tout $j \in [1, i-1]$ la j-ème création de nœud dans ajoute a lieu lors d'un stade $j+1$. Alors, on est dans le cas 1 pour tous les stades $< j$. Pour le stade j, on ne peut être au cas 1, car il implique que z vaut l'étape en cours, ce qui n'est possible que si la création de nœud au stade j a déjà eu lieu. Donc on est soit dans la cas 2, et le nœud est crée au stade j, soit dans le cas 3 et aucune création de nœud n'a lieu avant l'étape suivante. Dans les deux cas la propriété est vraie à l'étape $i+1$.

Donc la propriété est vérifiée par récurrence sur le nombre d'étapes.

De plus, ceci montre à quels moments intervient le cas 1. Et nous donne donc un moyen simple pour parcourir le moins d'états et de stades :



3.11. On construit une bijection entre les suffixes de $s\#$et les feuilles de l'arbre associé au mot $s\# = t$. Soit $t[j..|\mathtt{t}|]$ un suffixe de $s\#$. Au plus tard à la fin du stade j de l'étape $|\mathtt{t}|$ existe un nœud ν_j tel qu'il existe un chemin d'étiquette $t[j..|\mathtt{t}|]$ se terminant en position z du nœud ν_j. On a par définition $t[z] = \#$. Comme $\forall i,\ s[i] \neq \#$, on a donc nécessairement $z = fin(\nu_j) = |\mathtt{t}|$ et $fils(\nu_j) = []$: ν_j est bien une feuille.

Injectivité : Soit $i, j \in [1, |\mathtt{t}|]$ tels que $\nu_i = \nu_j$. Les chemins d'étiquettes $t[i..|\mathtt{t}|]$ et $t[j..|\mathtt{t}|]$ se terminent respectivement en positions z et z' du nœud $\nu_i = \nu_j$. Or on a montré ci-dessus $z = fin(\nu_i) = |\mathtt{t}| = fin(\nu_j) = z'$. Donc $z = z'$ et les chemins sont d'étiquettes de même longueur, c'est-à-dire : $i = j$, d'où l'injectivité.

Surjectivité : Soit ν une feuille de l'arbre. Par construction, on a $fin(\nu) = |\mathtt{t}|$. En effet,

- A est initialisé à $(1, 0, [(1, |\mathtt{t}|, [])])$ dont les feuilles se terminent en $|\mathtt{t}|$.
- Cela reste vrai lorsque :

$$fils(B) \leftarrow (i, |\mathtt{t}|, [])::fils(B)$$

  et lorsque :

$$fils(B) \leftarrow [(z + 1, fin(B), fils(B)), (i, |\mathtt{t}|, [])]$$

  dans ajoute(A,t,i,j).

D'où le résultat par récurrence sur le nombre d'appels à ajoute.

De plus, on a déjà montré que le chemin qui se termine en position $fin(\nu)$ du nœud ν a une étiquette qui est un sous-mot de t, celui-ci est donc de la forme $s[j..|\mathtt{t}|]$, $j \in [1, |\mathtt{t}|]$. Par unicité des chemins, on a donc $\nu = \nu_j$. D'où la surjectivité.

La bijection $t[i..|\mathtt{t}|] \mapsto \nu_i$ répond à la question.