Centrale Supélec 2010 — Corrigé

I. Quelques exemples

A. Cas des langages reconnaissables

1. Le test d'appartenance d'un mot m à un langage reconnaissable s'effectue en $O(|m|)$ comme l'illustre la fonction suivante :

   fonction **accepte** m
   | $l \leftarrow$ longueur(m) ;
   | $q \leftarrow q_i$;
   | $i \leftarrow 0$;
   | tant que $i < l$ faire
   | | $q \leftarrow \delta(q, m[i])$;
   | | $i \leftarrow i+1$;
   | fin tant que ;
   | retourner $q \in F$
   fin fonction

   La boucle est répétée $|m|$ fois et chaque itération s'exécute en temps constant. La fonction a donc une complexité temporelle en $O(|m|)$ et L est donc polynomial.

2. En conservant les notations précédentes, étant donné un automate $\mathscr{A}$ reconnaissant un langage L, on en déduit un automate $\mathscr{A}'$ reconnaissant $L^\star$ en ajoutant à $\mathscr{A}$ les transitions $p \xrightarrow{a} q$ pour chaque état $p \in F$ et chaque transition $q_i \xrightarrow{a} q$ ayant l'état initial pour origine.
   En effet, tout chemin réussi de $\mathscr{A}^\star$:

   $$q_i = p_0 \xrightarrow{a_0} p_1 \xrightarrow{a_1} p_2 \cdots p_{n-1} \xrightarrow{a_{n-1}} p_n$$

   peut se décomposer, en le « coupant » aux transitions que l'on vient d'ajouter, en un certain nombre de sous-chemins, chacun commençant dans $\{q_i\} \cup F$ et terminant dans F et étiquetté par un mot de L. L'étiquette du chemin précédent est donc une mot de $L^\star$. Réciproquement, à partir de chemins reconnaissant des mots de L, il est possible de déduire un chemin accepté par $\mathscr{A}^\star$ et reconnaissant un mot de $L^\star$.
   L'automate $\mathscr{A}^\star$ n'est pas forcément déterministe, mais cela n'est pas exigé par l'énoncé. Cependant, pour pouvoir reconnaître $L^\star$ en temps linéaire, il faut le déterminiser.

3. La description de la construction de $\mathscr{A}^\star$ à partir de $\mathscr{A}$ montre que l'on ajoute $|F| \times |A|$ transitions. Ainsi, cette construction a une complexité temporelle et spatiale en $O(|F| \times |A|)$ (on suppose que l'ajout d'une simple transition est en $O(1)$).
   À nouveau, pour reconnaître $L^\star$ en temps linéaire, il faut ensuite déterminiser $\mathscr{A}^\star$, opération qui a un coût exponentielle, mais qui ne dépend que du langage et non du mot que l'on souhaite tester.

B. Le cas de $L_0 = \{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbf{N}\}$

1. Le langage $L_0 = \left\{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbf{N}\right\}$ n'est pas reconnaissable. En effet, si c'était le cas, il existerait un entier n_0 tel que tout mot $u \in L_0$ de longueur au moins n_0 pourrait être décomposé sous la forme $u = x \cdot y \cdot z$ avec $1 \leq |y| \leq n_0$ et telle que pour tout $k \in \mathbf{N}$, le mot $x \cdot y^k \cdot z$ appartient à L_0.
   Pour $u = \mathtt{a}^{n_0}\mathtt{b}^{n_0}$, avec les notations précédentes, trois possibilités sont possibles :
   – Le mot y est de la forme $\mathtt{a}^\alpha$ avec $1 \leq \alpha \leq n_0$. Dans ce cas, on a $v = x \cdot y^2 \cdot z = \mathtt{a}^\alpha u$ et il est clair que v n'appartient ni à L_0, ni à $L_0^\star$;
   – Le mot y est de la forme $\mathtt{b}^\beta$. On traite ce cas comme le précédent ;
   – Le mot y est de la forme $\mathtt{a}^\alpha\mathtt{b}^\beta$ avec $1 \leq \alpha$, $1 \leq \beta$ et $\alpha + \beta \leq n_0$. On a en particulier $\alpha < n_0$ et $b < n_0$. Ains, le mot $v = x \cdot y^2 \cdot z = \mathtt{a}^{n_0}\mathtt{b}^\beta\mathtt{a}^\alpha\mathtt{b}^{n_0}$ n'est à nouveau ni dans L_0, ni même dans $L_0^\star$.
   En conclusion, ni L_0 ni $L_0^\star$ ne sont reconnaissables.

2. Un programme polynomial (et même linéaire) reconnaissant L_0 est le suivant. L'alphabet est représenté par le type `lettre` et les mots sont des listes de lettres :

```
type lettre = A | B ;;

let reconnait_L0 m =
  let rec lis_a compte = function
    [] -> compte = 0
  | A :: m' -> lis_a (compte + 1) m'
  | B :: m' -> if compte > 0 then lis_b (compte - 1) m' else false
  and lis_b compte = function
    [] -> compte = 0
  | B :: m' -> if compte > 0 then lis_b (compte - 1) m' else false
  | A :: m' -> false
  in
  lis_a 0 m
;;
```

   On adapte aisément ce programme pour reconnaître les mots de $L_0^\star$ en remplaçant la sous-fonction `lis_b` par la suivante :

```
...
and lis_b compte = function
  [] -> compte = 0
| B :: m' -> if compte > 0 then lis_b (compte - 1) m' else false
| A :: m' -> if compte = 0 then lis_a 1 m' else false
in ...
```

C. Un autre exemple

On note donc $A = \{\mathtt{a}\}$ et $L_1 = \left\{ \mathtt{a}^{2^n} \;\middle|\; n \in \mathbf{N} \right\}$.

1. Le langage L_1 n'est pas reconnaissable. En effet, par application du lemme de l'étoile, si il existe un entier n_0 tel que tout mot $u \in L_1$ tel que $|u| \geq n_0$ peut s'écrire sous la forme $x \cdot y \cdot z$ avec $1 \leq |y| \leq n_0$ et $x \cdot y^2 \cdot z$ (entre autres).

   Dans notre cas, soit p tel que $n_0 < 2^p$ et soit $u = \mathtt{a}^{2^p}$. En notant $y = \mathtt{a}^\alpha$ avec $1 \leq \alpha \leq n_0 < 2^p$, on aurait $\mathtt{a}^{2^p+\alpha} \in L_1$ ce qui est impossible puisque $2^p < 2^p + \alpha < 2^{p+1}$. Ainsi, L_1 n'est pas reconnaissable.

2. Le langage L_1 est par contre clairement polynomial. En effet, il suffit pour vérifier qu'un mot $\mathtt{a}^l$ est dans L_1 de montrer que sa longueur l est une puissance de 2.

   Il faut tout d'abord récupérer cette longueur. Cela peut être fait peut faire en temps constant si le mot est représenté par un tableau avec sa taille directement accessible, ou en temps $l \log_2 l$ si l'on doit parcourir toutes les lettres pour les compter. Dans le deuxième cas, on incrémente n fois un entier inférieur à n et donc dont la longueur en bit est au plus $1 + \log_2 l$.

   Ensuite, il faut vérifier que l'entier l obtenu est bien une puissance de 2. Il faut vérifier qu'il est non nul (car $\varepsilon \notin L_1$), et que soit $l = 1$, soit $l \equiv 0[2]$ et $l/2$ est pair.

   Un test de parité se fait en temps constant, une division par deux d'un entier se fait en $1+\log_2 l$ (il s'agit d'un simple décalage de bits), et on doit au plus tester $1+\log_2 l$ bits.

   On a donc décrit un algorithme qui teste si un mot m est dans L_1 en $O(|m|^2)$. Ainsi, L_1 est polynomial.

3. Comme $L_1^\star \subseteq A^\star$ et que $\mathtt{a} = \mathtt{a}^{2^0}$ appartient à $L_1^\star$, on en déduit que $L_1^\star = A^\star$.

4. On en déduit donc que $L_1^\star$ est polynomial et même reconnaissable.

D. Un dernier exemple

1. Le langage L_2 est bien polynomial. En effet, pour vérifier qu'un mot est dans L_2, il faut :

   (a) vérifier qu'il est de la forme $\varphi_1\#\varphi_2\#\varphi_2\#$ avec φ_1, φ_2 et φ_3 dans $\{0,1\}^\star$;

   (b) calculer les entiers n_1, n_2 et n_3 représentés respectivement par φ_1, φ_2 et φ_3 ;

   (c) vérifier que $n_1 n_2 = n_3$.

   Chaque étape se fait en temps polynomial : (a) en temps linéaire suivant $|m|$, pour (b) chaque calcul de n_i se fait au pire en $O(|\varphi_i|^2)$ et pour (c) on calcule le produit de deux entier en $O(|\varphi_1| \times |\varphi_2|)$ et on le compare à n_3 en $O(|\varphi_3|)$.

   Ainsi, le langage L_2 est bien polynomial.

2. Il n'est pas contre par reconnaissable. En effet, il n'est pas possible de découper un mot u de L_2 sous la forme $u = x \cdot y \cdot z$ tel que $y \neq \varepsilon$ et que $x \cdot z$ soit dans L_2. On vérifie ceci aisément en examinant tous les cas possibles pour y (en particulier, les cas intéressants sont suivant que y contient la lettre # ou non).

E. Une petite variation

1. Un mot de $\psi(L)$ est la concaténation d'un même mot de L. Comme $L^\star$ contient L et est stable par concaténation, on en déduit que $\psi(L)$ est inclut dans $L^\star$.

   Si $L = \{\mathtt{ab}, \mathtt{ba}\}$, on a $\mathtt{abba} \in L^\star$ mais $\mathtt{abba} \notin \psi(L)$. Ainsi, l'inclusion peut être stricte.

2. Définissons le langage $L = \left\{ \mathtt{ab}^k \;\middle|\; k \in \mathbf{N} \right\}$ sur l'alphabet $A = \{\mathtt{a}, \mathtt{b}\}$. Il est clairement reconnaissable, étant décrit par l'expression régulière $\mathtt{ab}^\star$. Par contre, $\psi(L)$ n'est pas reconnaissable. En effet, pour n assez grand, il n'est pas possible d'appliquer le lemme de l'étoile au mot $\mathtt{ab}^n$, ce que l'on vérifie en étudiant toutes les décompositions de ce mot sous la forme $x \cdot y \cdot z$ avec $y \neq \varepsilon$.

   Ainsi, « L est reconnaissable » n'implique pas que « $\psi(L)$ est reconnaissable ».

3. Si L est polynomial, alors $\psi(L)$ l'est aussi. En effet, si l'on considère une fonction f qui indique en temps polynomial si un mot est dans L ou non, étant donné un mot m, on teste pour tout k divisant $|m|$ si le préfixe p_k de m de longueur k est dans L et si $m = (p_k)^{|m|/k}$. Pour $1 \leq k \leq m$, le test de divisibilité se fait en temps polynomial (et même quadratique, puisque l'on calcule le reste de la division euclidienne en effectuant au plus $|m|$ soustractions) et le test d'égalité $m = (p_k)^{|m|/k}$ se fait en temps linéaire.

4. Cela découle de la question précédente, et du fait qu'un langage reconnaissable est polynomial.

II. Trois algorithmes

A. Une énumération des parties de $[\![0, n-1]\!]$

1. On peut imaginer la fonction suivante :

```
let decompose n k =
  let tab = make_vect n false in
  let r = ref k in
  for i = 0 to n - 1 do
    if !r mod 2 = 1 then tab.(i) <- true ;
    r := !r / 2
  done ;
  tab
;;
```

   Éventuellement, on ajoutera la vérification de $0 \leq k \leq 2^{n-1}$ en levant une exception à la fin de la boucle si `!r` n'est pas nul.

2. On peut à partir de la fonction précédente énumérer toutes les parties de $[\![0, n-1]\!]$ puisque pour $0 \leq k \leq 2^{n-1}$, la décomposition de k (et plus précisément la position des bits valant `true`) indique quels éléments de $[\![0, n-1]\!]$ il faut prendre. Toutes les parties de l'ensemble seront ainsi décrites, puisque il y en a 2^n.

3. Étant donné un mot m de longueur $p+2$, une décomposition de m en au moins 2 mots non vides peut être représenté par la position des dernières lettres de tous les sous-mots à part le dernier, ce qui constitue une partie non vide de $[\![0, p]\!]$.

   Ainsi, la décomposition de l'exemple est représentée par $\{3, 7\}$ (on a alors $p = 15$) :

```
                    0123 4567 8901234
centralesupelec = cent|rale|supelec
```

4. Pour répondre à cette question, on s'autorise à modifier légèrement le type de la fonction `dans_L`. En effet, pour simplifier le programme (et réduire la complexité), on supposera que cette fonction prendra en entrée un mot m représenté par un tableau, ainsi que des indices $0 \leq d \leq f \leq |m| - 1$ et que la fonction indiquera si le sous-mot de m partant de l'indice d (inclus) jusqu'à l'indice f (incluse lui aussi) est dans L ou non.

   On commence tout d'abord par définir une fonction qui, étant donné un tableau de booléen (comme celui obtenu par la fonction `decompose` précédente) de longueur n et un entier k, indique la position du premier `true` du tableau à partir de la position k incluse, ou n sinon.

```
let rec suivant tab n k =
  if k < n && not tab.(k) then suivant tab n (k + 1) else k ;;
```

Ensuite, on écrit la fonction `teste_decomposition` qui teste si une décomposition donnée découpe bien un mot en mots du langage L. En particulier, si m est de longueur l, le tableau représentant m correspondra aux indices allant de 0 à $l-1$ et donc la décomposition utilisée doit être une décomposition de $[\![0, l-2]\!]$.

```
let rec teste_decomposition dans_L mot decomp =
  let longueur = vect_length mot in
  let rec aux pos =
    if pos = longueur then true (* on a tout parcouru *)
    else
      let fin = suivant decomp (longueur - 1) pos in
      if dans_L mot pos fin then aux (fin + 1) else false
  in aux 0
;;
```

Enfin, étant donné un mot, on teste s'il est dans L ou si on peut le décomposer en mots de L. Pour tester les différentes sous-parties, on les énumérera comme expliqué précédemment, et on utilisera une exception dès que l'on aura trouvé une bonne décomposition. On a donc :

```
exception Eject ;;

let dans_L_etoile dans_L mot =
  let longueur = vect_length mot in
  if dans_L mot 0 (longueur - 1) then true else
  if longueur <= 1 then false (* pas de décomposition possible *)
  else try
    let borne = (puissance (longueur - 1)) - 1 in
    for i = 1 to borne do
      let decomp = decompose (longueur - 1) i in
      if teste_decomposition dans_L mot decomp then raise Eject
    done ;
    false
  with Eject -> true
;;
```

Pour calculer $2^{|m|-1}$, on utilise la fonction largement optimisable suivante :

```
let rec puissance k =
  if k = 0 then 1 else 2 * (puissance (k-1)) ;;
```

Cette fonction ne permet de traiter le mot vide, mais il semble que celui-ci soit traité séparément puisque l'on considère des mots non vides (que l'on décompose en concaténation de mots non vides).

5. Les fonctions `decompose` et `suivant` s'exécutent en temps linéaire.

   La fonction `teste_decomposition` effectue $O(n)$ opérations élémentaires, et autant d'appels à `dans_L` qu'il y a de `true` dans la décomposition (vis-à-vis de la complexité de la fonction `suivante`, on parcours l'intégralité de la décomposition une unique fois).

   La fonction `puissance` s'exécute elle aussi en temps linéaire en la longueur du mot.

   Ainsi, la complexité temporelle de `dans_L_etoile`, sans compter les appels à `dans_L`, est de l'ordre de $n2^{n-1}$ (puisqu'il y a 2^{n-1} parties à tester). Concernant le nombre d'appels à `dans_L`, il est égal au nombre de `true` apparaissant dans toutes les décompositions des entiers entre 1 et 2^{n-2}. Il y en a donc de l'ordre de $(n-1)\times 2^{n-2}$.

6. Pour éviter de manipuler de trop grands nombres, on peut énumérer les parties d'un ensemble à n éléments en imbriquant n boucles pour autant de compteurs allant de 0 (pour `false`) à 1 (pour `true`).

   Une autre possibilité est de faire l'énumération récursivement : les parties de $[\![0,n]\!]$ sont les parties de $[0,n-1]$ et ces mêmes parties auxquelles on a ajouté n.

   Dans les deux cas, on obtient une complexité comparable, puisqu'à chaque fois, l'élaboration de la décomposition se fait en temps linéaire.

B. Un algorithme récursif

1. On a :
$$L^\star = \bigcup_{k\geq 0} L^k = \{\varepsilon\} + \bigcup_{k\geq 1} L^k = \{\varepsilon\} + L\cdot\bigcup_{k\geq 0} L^k = \{\varepsilon\} + L\cdot L^\star$$

   En particulier, un mot non vide m de $L^\star$, s'il n'appartient pas à L, peut s'écrire sous la forme $m = x\cdot y$ avec $x\in L\setminus\{\varepsilon\}$ et $y\in L^\star\setminus\{\varepsilon\}$, ce qui correspond à la propriété demandée.

2. On peut donc écrire une nouvelle fonction d'appartenance à $L^\star$:

```
let dans_L_etoile_2 dans_L mot =
  let dernier = vect_length mot - 1 in
  let rec aux pos =
    if dans_L mot pos dernier then raise Eject else
    for i = pos to dernier - 1 do
      if dans_L mot pos i then aux (i + 1)
    done
  in
  try
    aux 0 ; false
  with Eject -> true
;;
```

3. Si $c(n)$ désigne le nombre d'appels à la fonction `dans_L` dans le pire des cas pour un mot de longueur n, on a $c(1) = 1$ et :
$$c(n+1) = 1 + \sum_{k=1}^{n} [1 + c(k)] = (n+1) + \sum_{k=1}^{n} c(k)$$

   En effet, le « $1+\ldots$ » correspond au test de l'appartenance de m à L, et la somme correspond aux n valeurs possibles prises par i dans la boucle.

   On a $c(1) = 1$, $c(2) = 2 + c(1) = 3$, $c(3) = 3 + c(1) + c(2) = 7$ et, plus généralement, on vérifie sans peine que l'on a pour tout $n\geq 1$, $c(n) = 2^n - 1$. Ce nombre d'appel est atteint lors que le mot donné n'appartient pas à $L^\star$.

C. Une programmation dynamique

1. Pour $i\in[\![0,n-1]\!]$, le facteur correspondant à $T_{i,i}$ n'étant pas décomposable puisqu'il ne contient qu'une lettre, il appartient à $L^\star$ si, et seulement si il appartient à L. On a donc, avec les notations précédentes :
$$T_{i,i} = \texttt{dans\_L mot i i}$$

2. D'après la question II.B.1), pour $i<j$, le facteur $m_i\ldots m_j$ (qui comporte au moins deux lettres) est dans $L^\star$ si, et seulement si, il est dans L ou bien on peut l'écrire comme la concaténation de deux mots non vide appartenant tous deux à $L^\star$.

   La relation demandée traduit directement cette équivalence, l'indice k indiquant où la césure doit avoir lieu.

3. Dans la formule précédente, les booléens $T_{i,k}$ et $T_{k+1,j}$ correspondent à des facteurs de longueur strictement inférieure au facteur correspondant à $T_{i,j}$.

   On peut donc en déduire un algorithme qui décide si un mot m est dans $L^\star$ ou non : on définit un tableau de booléen qui indiquera, à la fin de l'algorithme, pour chaque facteur de m, s'il est dans $L^\star$ ou non. La remarque précédente indique que l'on peut remplir simplement ce tableau en procédant par taille croissante de sous-mot.

   À la fin, il suffit de retourner la valeur de $T_{0,|m|-1}$.

4. Cela se traduit par le programme suivant :

```
let dans_L_etoile_3 dans_L mot =
  let longueur = vect_length mot in

  (* on commence par déclarer le(s) tableau(x) qui
```

```
    correspondant aux différentes T_{i,j} *)
  let tab = make_vect longueur (make_vect 1 false) in
  for i = 0 to longueur - 2 do
    tab.(i) <- make_vect (longueur - i) false
  done ;

  (* on initialise les T_{i,i} *)
  for i = 0 to longueur - 1 do
    tab.(0).(i) <- dans_L mot i i
  done ;

  (* on remplit maintenant le reste du tableau *)
  for i = 2 to longueur do (* i désigne la longueur de mot *)
    for j = 0 to longueur - i do
    (* j désigne la position de départ *)
      (* on s'intéresse donc à T_{j, j + i - 1} *)
      if dans_L mot j (j + i - 1)
      then tab.(i - 1).(j) <- true
      else
        for k = j to i + j - 2 do
          (* on regarde T_{j, k} puis T_{k + 1, j + i - 1} *)
          if tab.(k - j).(j) && tab.(i + j - k - 2).(k + 1) then
            tab.(i - 1).(j) <- true
        done
    done
  done ;
  tab.(longueur - 1).(0)
;;
```

5. La complexité s'évalue de la façon suivante :
   – l'initialisation se fait en $O(n^2)$;
   – le remplissage de `tab.(0)` se fait en $O(n)$ plus $O(n)$ appels à `dans_L` ;
   – le remplissage de la ligne i se fait en répétant $(n-i+1)$ fois la boucle de j qui consiste en un appel à `dans_L` plus des opérations en temps constant suivi de, dans la boucle pour k repétée $i-1$ fois, des opérations en temps constant ;
   – la sélection du résultat retourné, en temps constant.

   En conclusion, la fonction effectue $O(n^2)$ appels à `dans_L` et des opérations élémentaires en $O(n^3)$.
6. La complexité spatiale de cette solution est en $O(n^2)$, du fait de la déclaration du tableau qui contient tous les calculs intermédiaires. Par contre, la complexité temporelle est polynomiale. Ainsi, si les deux algorithmes précédents ont une complexité spatiale linéaire (du fait de la pile d'appels récursifs ainsi que du stockage de la décomposition), leur complexité est exponentielle pour `dans_L_etoile2` voire pire, soit $O(n2^n)$ pour `dans_L_etoile` ce quie signifie qu'il n'est pas utilisable en pratique.

III. Utilisation d'un graphe

A. Structure de file

1. On a fait l'hypothèse que le nombre total d'entrées reste inférieure à la taille du tableau, on n'a donc pas besoin de tests de dépassement pour `put`. De même, la présence d'une fonction `est_vide` permet de supposer que la file est non vide lors d'un appel à `get`.

   Les fonctions demandées peuvent être écrites ainsi :

```
let put entier file =
  let fin' = file.fin + 1 in
  file.contenu.(fin') <- entier ;
  file.fin <- fin'
;;

let get file =
  let debut = file.debut in
  file.debut <- file.debut + 1 ;
  file.contenu.(debut)
;;
```

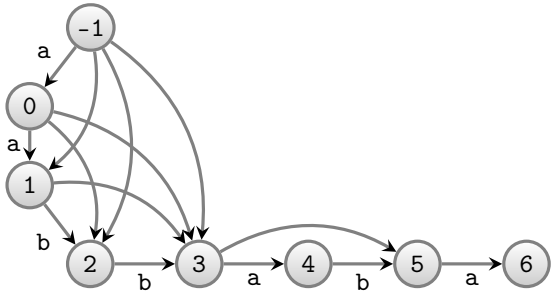
2. Si l'on suppose que le nombre total d'éléments est plus grand que la taille du tableau, mais que celle-ci n'est jamais excédée par le nombre d'éléments restants dans la file à un moment donné, il est possible de « réutiliser » des cellules du tableau en le voyant de manière cylindrique, la cellule d'indice `0` succédant la dernière cellule du tableau.

   Pour cela, l'incrémentation d'un indice se fera modulo longueur du tableau. Par exemple, dans la fonction `put`, la définition de `fin'` devient :

```
...
let fin' = (file.fin + 1) mod (vect_length file.contenu) in
...
```

B. Introduction d'un graphe orienté

1. Le graphe $\mathcal{G}_{L_0}$(aabbaba) est le suivant (on n'a laissé les étiquettes que pour les mots de une lettre pour augmenter la lisibilité) :



2. On peut déduire l'algorithme suivant :
   – Tout d'abord, on définit le graphe en allouant puis en remplissant la matrice d'adjacence correspondante ;
   – Ensuite, on exécute l'algorithme de recherche de chemin à l'aide d'un parcours en largeur comme décrit dans l'énoncé.

   Pour que le tableau alloué soit de plus petite taille possible, l'information concernant une flèche allant de i à j correspond à la cellule `g.(j).(i)`.

```
let dans_L_etoile_4 dans_L mot =
  let longueur = vect_length mot in

  (* on alloue la matrice d'adjacence *)
  let g = make_vect longueur [| false |] in
  for i = 1 to longueur - 1 do
    g.(i) <- make_vect (i + 1) false
  done ;

  (* on la remplie ensuite à l'aide de dans_L *)
  for i = 0 to longueur - 1 do
    for j = i to longueur - 1 do
      g.(j).(i) <- dans_L mot i j
    done
  done ;

  (* on initialise la file à la bonne taille et le tableau vu *)
  let file = creer_file longueur
  and vus = make_vect longueur false in

  (* algorithme de parcours en largeur *)
  put (-1) file ;
  while not (est_vide file) do
    let s = get file in
    for i = s + 1 to longueur - 1 do
      (* T_{s+1...i} *)
      if not vus.(i) && g.(i).(s + 1) then begin
        vus.(i) <- true ;
        put i file
      end
    done
  done ;

  (* on retourne le bon résultat *)
  vus.(longueur - 1)
;;
```

3. Pour l'initialisation des tableaux, le nombre d'appels à `dans_L` et le nombre d'opérations élémentaires sont tous deux en $O(n^2)$. Pour le parcours en profondeur, chaque sommet sera inséré au plus une fois dans la file, et pour chaque sommet, on a au plus n arêtes à inspecter (en fait, $n-i$ pour le i-ème sommet). Donc le parcours en profondeur se fait dans le pire des cas en $O(n^2)$.
4. On peut résumer l'analyse des différents algorithmes par le tableau suivant :

Algorithme	Complexité spatiale	Appels à `dans_L`	Opérations élémentaires
`dans_L_etoile`	$O(n)$	$O(n2^n)$	$O(n2^n)$
`dans_L_etoile_2`	$O(n)$	$O(2^n)$	$O(2^n)$
`dans_L_etoile_3`	$O(n^2)$	$O(n^2)$	$O(n^3)$
`dans_L_etoile_4`	$O(n^2)$	$O(n^2)$	$O(n^2)$

Si les deux premiers algorithmes ont une complexité spatiale meilleure, leur complexité temporelle exponentielle les rends difficilement utilisables en pratique.

Concernant les deux derniers algorithmes, la complexité spatiale quadratique n'est pas réellement un problème, sachant qu'en contrepartie on obtient une complexité temporelle polynomiale (en supposant que `dans_L` l'est). En particulier, on a obtenu une preuve constructive du fait que si un langage est polynomial, alors son étoile l'est aussi.

A priori, l'algorithme 4 est plus rapide que l'algorithme 3. Cependant, à moins d'avoir une fonction `dans_L` qui n'est pas au moins $\Theta(n)$, c'est la partie concernant les appels à `dans_L` qui va imposer la classe de complexité à `dans_L_etoile`, et les algorithmes 3 et 4 seront alors dans la même classe.

Notons pour finir que dans le cas d'un langage reconnaissable, la décision de l'appartenance à $L^\star$ se fait en temps linéaire, car l'utilisation de l'automate reconnaissant L permet de déterminer exactement à quels endroits les césures pourront être effectuées.