ÉCOLES NORMALES SUPÉRIEURES

CONCOURS D'ADMISSION 2017 FILIÈRE MP – CONCOURS INFO

COMPOSITION D'INFORMATIQUE-MATHÉMATIQUES – (ULCR)

(Durée : 4 heures)

L'utilisation des calculatrices n'est pas autorisée pour cette épreuve.

Ce sujet comprend 10 pages numérotées de 1 à 10.

⋆ ⋆ ⋆

Détection de carrés dans les mots

Ce sujet traite de la recherche de répétitions dans un texte. Plus particulièrement, on s'intéresse aux répétitions appelées **carrés**.

La partie I introduit le problème de la recherche de répétitions dans un texte et donne un premier algorithme permettant de le résoudre. La partie II est consacrée à la construction d'un mot infini sans carré. Enfin, les parties III et IV sont consacrées à la mise au point d'un algorithme efficace de détection de carrés.

La **complexité**, ou le **coût**, d'un programme P est le nombre d'opérations élémentaires (addition, soustraction, affectation, test, lecture ou écriture dans un tableau, etc...) nécessaires à l'exécution de P dans le cas le pire. Lorsque cette complexité dépend d'un ou plusieurs paramètres $n_1, \ldots, n_k$, on dira que P **a une complexité en** $\mathcal{O}(f(n_1, \ldots, n_k))$, s'il existe une constante $K > 0$ telle que, pour toutes les valeurs de $n_1, \ldots, n_k$ suffisamment grandes (c'est-à-dire plus grandes qu'un certain seuil), pour toute instance du problème de paramètres $n_1, \ldots, n_k$, la complexité de P est au plus $Kf(n_1, \ldots, n_k)$. Lorsqu'il est demandé de garantir une certaine complexité, le candidat devra justifier cette dernière.

On considère un **alphabet** Σ, c'est-à-dire un ensemble fini d'éléments appelés **lettres**. On appelle Σ^* l'ensemble de tous les mots de longueur finie utilisant les lettres de l'alphabet Σ. La **longueur** d'un mot x, c'est-à-dire le nombre de lettres qui le composent, est notée $|x|$. Le **mot vide**, de longueur 0 et noté ε, appartient également à Σ^*. Le mot obtenu par **concaténation** de deux mots x et y, noté xy, a pour longueur $|x| + |y|$ et est composé des lettres de x, suivies des lettres de y.

Un **carré** est un mot égal à xx où x est un mot non vide. On dit qu'un mot m **contient un carré** s'il existe un mot non vide x et deux mots (éventuellement vides) y et z tels que :

$$m = yxxz$$

On dit alors que xx est un **carré de** m. La **période** d'un carré xx est la longueur du mot x. Par exemple, le mot *bonbon* est un carré de période 3, et le mot *repetition* contient le carré *titi* de période 2.

On choisit de représenter les mots par des tableaux de caractères. Les cases d'un tableau de longueur n sont numérotées de 0 à $n - 1$. Pour des raisons de cohérence entre les notations mathématiques et les représentations en machine, on note x_0, x_1, …, $x_{|x|-1}$ les lettres qui composent le mot x de Σ^*. On a donc pour tout mot non vide x :

$$x = x_0 \dots x_{|x|-1}$$

On dit aussi que x_i est la lettre qui apparaît à la **position** i **du mot** x. La numérotation des lettres d'un mot permet de parler de la **position à laquelle un carré apparaît dans un mot**. Par exemple, le mot *tintinnabuler* est représenté par le tableau décrit ci-dessous.

Numéro de la case	0	1	2	3	4	5	6	7	8	9	10	11	12
Caractère contenu dans la case	*t*	*i*	*n*	*t*	*i*	*n*	*n*	*a*	*b*	*u*	*l*	*e*	*r*

On dit que le mot (ou le tableau qui le représente) contient le carré *tintin* de période 3, à la position 0. En effet, la première lettre de *tintin* apparaît à la case 0 du tableau. Le mot contient aussi le carré *nn* de période 1, à la position 5.

Le sujet de ce devoir s'appuie sur le langage informatique Caml Light. Le candidat pourra rédiger ses réponses dans ce langage, ou dans un format pseudo-code proche. On rappelle quelques fonctions Caml Light pour manipuler les tableaux.

— `vect_length m` renvoie le nombre de cases du tableau `m`.
— `sub_vect m x y` renvoie le sous-tableau de `m` constitué des `y` cases à partir de la case `x`.
— `make_vect x y` renvoie un tableau de `x` cases qui contiennent toutes la valeur `y`.

```
(* Caml *)  vect_length : 'a vect -> int
(* Caml *)  sub_vect : 'a vect -> int -> int -> 'a vect
(* Caml *)  make_vect : int -> 'a -> 'a vect
```

Partie I. Existence d'un carré

Question 1 On considère le programme suivant :

```
let carre_pe_pos m p i =
  let j = ref 0 in
    while  !j < p &&
           i + !j + p < (vect_length m) &&
           m.(i + !j + p) = m.(i + !j)
    do
      j := !j + 1
    done;
    !j;;
```

`(* Caml *)  carre_pe_pos : char vect -> int -> int -> int`

Donner les valeurs des appels suivants.

```
carre_pe_pos [|'c'; 'o'; 'u'; 'c'; 'o'; 'u'  |] 3 0;;
carre_pe_pos [|'c'; 'o'; 'u'; 'c'; 'o'; 'o' |] 3 0;;
carre_pe_pos [|'a'; 'b'; 'a'; 'b'; 'a'; 'b'; 'a'; 'b' |] 2 4;;
carre_pe_pos [|'a'; 'b'; 'a'; 'b'; 'a'; 'b'; 'a'; 'b' |] 2 5;;
```

Question 2 On suppose $p > 0$ et $i \geq 0$. Définir précisément ce que renvoie la fonction `carre_pe_pos` en fonction de ses arguments m, p et i. Que signifie le fait que la valeur renvoyée est égale à p ?

Question 3 Écrire une fonction `carre_pe` qui prend en entrée un mot m et un entier $p > 0$, et qui renvoie `true` si le mot m contient un carré de période p et `false` sinon.

`(* Caml *)  carre_pe : char vect -> int -> bool`

Question 4 Écrire une fonction `carre_naif` qui prend en entrée un mot m et renvoie `true` si le mot contient un carré de longueur quelconque, et `false` sinon.

`(* Caml *)  carre_naif : char vect -> bool`

Question 5 Dans cette question, on suppose que pour l'alphabet considéré, pour tout entier $\ell \geq 1$, il existe des mots de longueurs ℓ sans carré. Donner alors la complexité dans le pire des cas de la fonction `carre_naif` définie précédemment.

Question 6 On considère l'alphabet $\Sigma = \{a, b\}$. Montrer que tout mot suffisamment long contient un carré. Donner un algorithme efficace (s'exécutant en temps constant) permettant de décider l'existence d'un carré pour les mots m construits sur l'alphabet $\Sigma = \{a, b\}$. On pourra supposer ici que la fonction `vect_length` s'exécute en temps constant.

Partie II. Construction d'un mot infini sans carré

Le but de cette partie est de construire un mot sur un alphabet de quatre lettres, de longueur infinie, et sans carré. La notion de carré s'étend naturellement aux mots infinis. Un mot infini est une suite de lettres indicées par les entiers de $\mathbb{N}$. Le mot obtenu par **concaténation** d'un mot fini x et d'un mot infini y est le mot infini xy composé des lettres de x, suivies des lettres de y. Dans ce contexte, un **carré** reste un mot fini de la forme xx, avec x un mot non vide. Un mot infini m **contient un carré** s'il existe un mot fini non vide x, un mot fini y (éventuellement vide) et un mot infini z tels que $m = yxxz$. Un **bit** est un nombre entier égal à 0 ou 1. Soit $(b_i)_{i\in\mathbb{N}}$ une suite de bits tous nuls à partir d'un certain rang (c'est-à-dire qu'il existe N tel que pour tout $i \geq N$, $b_i = 0$). On dit que $(b_i)_{i\in\mathbb{N}}$ est la **représentation binaire** d'un nombre entier naturel n si

$$n = \sum_{i=0}^{+\infty} b_i 2^i.$$

On rappelle que tout nombre entier naturel possède une unique représentation binaire. On rappelle qu'en pratique, pour donner la représentation binaire d'un nombre entier dont les bits sont tous nuls à partir du rang N, on écrit simplement $b_{N-1}b_{N-2}\dots b_0$.

Soit n un nombre entier naturel et $(b_i)_{i\in\mathbb{N}}$ sa représentation binaire. On note $p(n)$ la **profondeur** de n, à savoir le plus petit entier i tel que $b_i = 0$. On note $c(n) = b_{p(n)+1}$, le $(p(n)+1)$-ème bit de la représentation binaire de n. On étudie le mot infini M sur l'alphabet $\{0, 1\}$ défini par

$$M = c(0)c(1)c(2)c(3)\dots$$

Question 7 Donner les huit premières lettres de M et vérifier que M contient des carrés de période 1 et 3.

Question 8 Soient $d \geq 0$ et $k \geq 1$ deux nombres entiers. On définit $i(d, k)$ comme l'unique nombre entier de l'ensemble $\{d, d+1, \dots, d+2^k-1\}$ égal à $2^{k-1}-1$ modulo 2^k. Soit n un nombre entier naturel. En comparant les représentations binaires de n et de n modulo 2^k, montrer que $p(i(d, k)) = k - 1$.

Question 9 Montrer que pour tous entiers $d \geq 0$ et $k \geq 1$,

$$c(i(d, k) + 2^k) = c(i(d, k)) + 1 \mod 2.$$

Question 10 Montrer que le mot M ne contient aucun carré de période p avec $p = 2^k$ et $k \geq 1$.

Question 11 Généraliser le raisonnement précédent pour en déduire que le mot M ne contient aucun carré de période paire.

Question 12 Définir un alphabet sur quatre lettres et donner, sur cet alphabet, un mot infini, c'est-à-dire une suite de lettres indicée par les entiers de $\mathbb{N}$, qui ne contient aucun carré.

Il existe également des mots infinis sur trois lettres sans carré, mais leur construction n'est pas demandée dans ce devoir. Dans la suite, on ne considère que des mots finis.

Partie III. Recherche efficace des carrés

Nous souhaitons mettre en place un algorithme efficace permettant la recherche de carrés dans un mot. Dans cette partie, nous nous intéressons à la détection de carrés créés lors de la concaténation de deux mots. Il s'en déduit un algorithme de test d'existence d'un carré dans un mot par la stratégie "diviser pour régner".

À titre d'exemple, considérons les mots $u = tin$ et $v = tinnabuler$. Le carré *tintin* apparaissant au début du mot uv est créé lors de la concaténation des mots u et v alors que le carré nn apparaissant dans le mot uv n'est pas créé lors de la concaténation de u et de v. Il apparaissait déjà dans le mot v.

Un mot y est un **préfixe** d'un mot x s'il existe un mot z tel que $x = yz$. Un mot y est un **suffixe** d'un mot x s'il existe un mot z tel que $x = zy$. Noter que le mot vide est préfixe et suffixe de n'importe quel mot.

Pour tous mots u et v sur un alphabet Σ et tout indice i tel que $0 \leq i \leq |v| - 1$, on définit :

1. $lms(u, v, i)$ est la longueur maximum d'un suffixe de u qui apparaît dans v en se terminant à la position i de v. Par exemple, $lms(lebon, dubonnet, 4) = 3$ parce que le suffixe *bon* de *lebon*, de longueur 3, apparaît dans *dubonnet* et se termine à la position 4 de *dubonnet*.
2. $lmp(u, v, i)$ est la longueur maximum d'un préfixe de u qui apparaît dans v en commençant à la position i de v. Par exemple, $lmp(pabon, pabonpapa, 5) = 2$ car le préfixe *pa* de *pabon* de longueur 2 apparaît à la position 5 de *pabonpapa*.

Question 13 On considère $\Sigma = \{a, b, c\}$. Soient $u = cabacbab$ et $v = cbacbabcbab$. Calculer $lms(u, v, i)$ et $lmp(v, v, i)$ pour i compris entre 0 et 10. Faites bien attention à lire correctement ce qui est demandé : pour *lms*, on demande u et v, mais pour *lmp*, on demande v et v. Donner votre réponse en recopiant et complétant le tableau suivant.

i	0	1	2	3	4	5	6	7	8	9	10
v_i	c	b	a	c	b	a	b	c	b	a	b
$lms(u, v, i)$											
$lmp(v, v, i)$											

Soient u et v deux mots. On appelle **nouveau carré pour la concaténation de u et v**, ou simplement **nouveau carré** quand il n'y a pas de risque de confusion, tout carré de uv qui est la concaténation d'un suffixe non-vide de u et d'un préfixe non-vide de v (de manière plus imagée, le carré est à cheval sur les deux mots, ou est créé par la concaténation de u et v).

Un nouveau carré ww qui apparaît en position i sur le mot uv est un **carré centré sur** u lorsque $i + |w| < |u|$. Si $i + |w| > |u|$ alors on dit qu'il est **centré sur** v. Par exemple, avec les mots u et v définis ci-dessus, *cbabcbacbabcba* est un nouveau carré de période 7 qui apparaît en position 4 sur le mot uv. Il est centré sur v. On remarquera que si $i + |w| = |u|$, alors ww est un nouveau carré qui n'est centré ni sur u, ni sur v.

Le schéma ci-dessous montre le cas général d'un nouveau carré obtenu lors de la concaténation de u et de v, centré sur v, de période p et se terminant à la position i de v.

$$u_0 \dots u_{|u|-2p+i} \underbrace{u_{|u|-2p+i+1} \dots u_{|u|-1} v_0 \dots v_{i-p}}_{\text{mot } w} \underbrace{v_{i-p+1} \dots \dots \dots \dots v_i}_{\text{mot } w} v_{i+1} \dots v_{|v|-1}$$

Question 14 Dans l'hypothèse où il existe dans uv un nouveau carré ww centré sur v, de période p et se terminant à la position i de v, calculer en fonction de p l'indice j tel que :

$$v_0 \dots v_{i-p} = v_{j+1} \dots v_i$$

Que peut-on dire des mots $u_{|u|-2p+i+1} \dots u_{|u|-1}$ et $v_{i-p+1} \dots v_j$?

Question 15 Montrer qu'il existe un carré de période p dans uv se terminant à la position i de v et centré sur v si et seulement si

(a) $1 \leq p < |v|$, et

(b) $p \leq i < 2p - 1$, et

(c) $2p - lms(u, v, p-1) - 1 \leq i \leq p + lmp(v, v, p) - 1$.

Question 16 Donner sans preuve une condition nécessaire et suffisante similaire à celle de la question précédente pour les carrés centrés sur u. On commencera par écrire le schéma montrant le cas général d'un nouveau carré centré sur u.

Question 17 Utiliser les résultats des questions précédentes pour donner les nouveaux carrés de période 7 centrés sur v obtenus lors de la concaténation de $u = cabacbab$ et $v = cbacbabcbab$.

Dans cette partie, on suppose donnée une implémentation des fonctions *lms* et *lmp*. En fait, pour des raisons d'efficacité du calcul, on ne dispose pas directement des fonctions *lms* et *lmp*, mais plutôt des fonctions suivantes :

```
(* Caml *)  calcul_lms : char vect -> char vect -> int vect
(* Caml *)  calcul_lmp : char vect -> char vect -> int vect
```

La fonction `calcul_lms` appliquée à u et v renvoie un tableau d'entiers de taille $|v|$ tel que l'entier stocké à la position i soit égal à $lms(u, v, i)$ pour $0 \leq i < |v|$. De façon similaire, la fonction `calcul_lmp` appliquée à u et v renvoie un tableau d'entiers de taille $|v|$ tel que l'entier stocké à la position i soit égal à $lmp(u, v, i)$ pour $0 \leq i < |v|$. On supposera que l'appel à chacune de ces fonctions sur les mots u et v a un coût $\mathcal{O}(|u| + |v|)$.

Question 18 Écrire une fonction `nouveau_carre_v` qui prend en entrée deux mots u et v et renvoie `true` si le mot obtenu lors de la concaténation de u et de v fait apparaître un nouveau carré centré sur v, et `false` sinon. On garantira une complexité en $\mathcal{O}(|u| + |v|)$.

```
(* Caml *)  nouveau_carre_v : char vect -> char vect -> bool
```

On admettra l'existence d'une fonction `nouveau_carre_u` (similaire à celle de la question précédente) de complexité $\mathcal{O}(|u|+|v|)$ qui renvoie `true` si un nouveau carré centré sur u apparaît lors de la concaténation des mots u et v (et renvoie `false` sinon).

Question 19 Écrire une fonction `nouveau_carre` qui prend en entrée deux mots u et v et renvoie `true` si un nouveau carré apparaît lors de la concaténation des mots u et v, et renvoie `false` sinon. On veillera à bien prendre en compte d'éventuels nouveaux carrés qui ne sont ni centrés sur u ni centrés sur v, et on garantira une complexité en $\mathcal{O}(|u| + |v|)$.

```
(* Caml *)  nouveau_carre : char vect -> char vect -> bool
```

Question 20 En déduire une fonction `carre` qui renvoie `true` si le mot m donné en entrée admet un carré, et `false` sinon. On garantira que le calcul se fait en $\mathcal{O}(|m| \times log(|m|))$.

```
(* Caml *)  carre : char vect -> bool
```

Partie IV. Implémentation efficace des fonctions *lms* et *lmp*

On souhaite maintenant implémenter efficacement le calcul de $lms(u, v, i)$, et $lmp(u, v, i)$. Comme annoncé dans la partie précédente, on implémente la fonction `calcul_lms` (respectivement `calcul_lmp`) qui, appliquée à u et v, renvoie un tableau d'entiers de taille $|v|$ tel que l'entier stocké dans la case i est égal à $lms(u, v, i)$ (respectivement $lmp(u, v, i)$) pour $0 \leq i < |v|$. On souhaite de plus garantir une complexité $\mathcal{O}(|u| + |v|)$ lors de l'appel à `calcul_lms u v` (respectivement `calcul_lmp u v`).

Pour programmer `calcul_lmp`, on met à jour itérativement la table des préfixes d'un mot comme expliqué ci-après. Considérons un mot v et pour tout entier $0 \leq i < |v|$, posons

$$pref_i = lmp(v, v, i)$$

Une méthode naïve pour calculer $pref_i$ consisterait à évaluer chaque valeur indépendamment des valeurs précédentes par comparaisons directes. Cependant, l'utilisation des valeurs déjà calculées permet d'obtenir un algorithme plus efficace.

Illustrons le procédé sur un exemple. Considérons le mot $v = aabaabaaab$, et supposons $pref_i$ déjà connu pour i allant de 0 à 3 comme indiqué dans la table des préfixes donnée ci-dessous.

i	0	1	2	3	4	5	6	7	8	9
v_i	a	a	b	$\boxed{a}$	$\boxed{a}$	$\boxed{b}$	$\boxed{a}$	$\boxed{a}$	a	b
$pref_i$	10	1	0	5	?	?	?	?	?	?

TABLE 1 – Table des préfixes pour le mot $v = aabaabaaab$.

Posons $u = v_3v_4v_5v_6v_7$. Le calcul de $pref_3$ nous dit que u est un préfixe de v (et c'est le plus long débutant à la position 3). Autrement dit, $u = v_0v_1v_2v_3v_4$ et $v_8 \neq v_5$.

1. On souhaite calculer $pref_4$. De l'égalité précédente, on déduit $v_4v_5v_6v_7 = v_1v_2v_3v_4$. Dans la mesure où $pref_1 \leq 4$, la situation à la position 4 est semblable à celle de la position 1. On a donc $pref_4 = pref_1 = 1$. De même, on a $pref_5 = pref_2 = 0$.
2. On regarde maintenant comment calculer $pref_6$. Du calcul de $pref_3$, on a $u = v_3v_4v_5v_6v_7$ préfixe de v, et donc les égalités suivantes : $v_3v_4 = v_0v_1$, $v_5 = v_2$, et $v_6v_7 = v_3v_4$. De plus, on a vu que $v_8 \neq v_5$. On en déduit que $pref_6 = 2$.
3. Enfin, regardons comment calculer $pref_7$. On a v_7 suffixe de u et préfixe de $v_4 \dots v_9$ et donc de v (car $pref_4 = 1$). On en déduit que $pref_7 \geq 1$. Pour savoir la longueur maximale du préfixe de v qui démarre à la position 7, il nous faut donc reprendre les comparaisons : v_8 contre v_1, v_9 contre v_2, ... On obtient ici $pref_7 = 3$.

Pour faire ce raisonnement de façon systématique, on introduit, l'indice $i \geq 2$ étant fixé, deux valeurs g et f qui constituent les éléments clés de la méthode. Elles satisfont les relations :

$$g = max\{j + pref_j \mid 0 < j < i\} \qquad f \in \{j \mid 0 < j < i \text{ et } j + pref_j = g\}$$

Question 21 Compléter la table 1 en indiquant la valeur de g et les valeurs possibles de f pour chaque indice i compris entre 2 et $|v| - 1 = 9$.

Question 22 Soient $1 < i < |v|$, $g = max\{j + pref_j \mid 0 < j < i\}$, et $f \in \{j \mid 0 < j < i \text{ et } j + pref_j = g\}$.

a. Que peut-on dire du mot $v_f \dots v_{g-1}$?

b. Lorsque que $g < i$, exprimer g en fonction de i.

Question 23 Soient $1 < i < |v|$, $g = max\{j + pref_j \mid 0 < j < i\}$, et $f \in \{j \mid 0 < j < i \text{ et } j + pref_j = g\}$. On suppose de plus que $i < g$. Montrer le résultat suivant :

$$pref_i = \begin{cases} pref_{i-f} & \text{si } pref_{i-f} < g - i \\ g - i & \text{si } pref_{i-f} > g - i \\ g - i + \ell & \text{sinon} \end{cases}$$

où ℓ est la longueur du plus long préfixe commun à $v_{g-i} \dots v_{m-1}$ et $v_g \dots v_{m-1}$. Autrement dit, on a $\ell = lmp(v_{g-i} \dots v_{m-1}, v_g \dots v_{m-1}, 0)$. On posera $k = pref_{i-f}$ et $k' = g - i$, et on étudiera séparément les cas $k < k'$, $k > k'$, et $k = k'$. Ces trois cas correspondent aux trois situations illustrées au travers des exemples donnés en début de cette partie.

Question 24 On considère le code donné en Figure 1.

```
 1. let calcul_pref v =
 2.   let pref = make_vect (vect_length v) 0 in
 3.    pref.(0) <- vect_length v;
 4.    let g = ref 0 in
 5.    let f = ref 0 in
 6.      for  i = 1 to  vect_length v - 1 do
 7.        if i < !g && pref.(i - !f) < !g - i then
 8.           pref.(i) <- pref.(i - !f)
 9.        else
10.          if i < !g && pref.(i - !f) > !g - i then
11.             pref.(i) <- !g - i
12.          else
13.           begin
14.             f:=i;
15.             g:= max !g i;
16.             while !g < vect_length v && v.(!g) == v.(!g - !f) do
17.               g := !g + 1;
18.             done;
19.             pref.(i) <- !g - !f;
20.           end;
21.      done;
22.      pref;;
```

FIGURE 1 – Code de la question 24.

```
(* Caml *)  calcul_pref : char vect -> int vect
```

a. Justifier que ce code réalise bien le calcul demandé, c'est-à-dire que la fonction `calcul_pref`, appliquée à un mot v, renvoie le tableau *pref* défini ci-dessus.

b. Justifier la terminaison de l'algorithme. Donner et justifier sa complexité.

Dans les trois questions suivantes, on demande des programmes courts et simples. En particulier, dans les deux questions suivantes, l'idée est d'utiliser la fonction `calcul_pref` sur des mots facilement obtenus à partir de u, v et d'un caractère spécial `'#'` n'appartenant pas à l'alphabet (on pourra utiliser des concaténations et des retournements).

Question 25 Donner une implémentation de complexité $\mathcal{O}(|u| + |v|)$ pour la fonction `calcul_lmp`.

Question 26 Afin d'implémenter la fonction `calcul_lms` demandée à la question suivante, on se propose tout d'abord de programmer une fonction `calcul_suff`, telle que (`calcul_suff v`) renvoie la table des suffixes du mot `v`, c'est-à-dire un tableau d'entiers de taille $|v|$ tel que l'entier stocké dans la case i est égal à $lms(v, v, i)$. Écrire le code de la fonction `calcul_suff v` en garantissant une complexité $\mathcal{O}(|v|)$. Attention, ici, on ne suppose pas disposer d'une implémentation

de la fonction `calcul_lms` car cette fonction sera implémentée à la question suivante à l'aide de `calcul_suff`.

```
(* Caml *)  calcul_suff : char vect -> int vect
```

Question 27 En utilisant la fonction `calcul_suff` de la question précédente, écrire le code de la fonction `calcul_lms`. On garantira une complexité en $O(|u| + |v|)$.

```
(* Caml *)  calcul_lms : char vect -> char vect -> int vect
```

Notes : Il existe de nombreuses constructions de mots infinis sans carrés. Le plus ancien est le mot de Thue. Le mot présenté dans ce sujet est original, mais très proche du mot de Dean [R.A. Dean, A sequence without repeats on x, x^{-1}, y, y^{-1}, *American Mathematical Monthly*, volume 72, pages 383–385, 1965]. L'algorithme présenté de ce sujet est dû à Main et Lorentz [M.G. Main and R.J. Lorentz, An $O(n \log n)$ algorithm for finding all repetitions in a string, *Journal of Algorithm*, volume 5, pages 422–432, 1984].

* *
*