ULC 521

INFORMATIQUE

(Durée : 4 heures)

Session 2005, Filière MP (groupes MPI et I) et PC (groupe I)

On étudie dans ce problème l'ordre lexicographique pour les mots sur un alphabet fini et plusieurs constructions des cycles de De Bruijn. Les trois parties sont largement indépendantes.

Définitions

– Dans tout le problème, $\mathcal{A}$ est un alphabet fini, de cardinal k, muni d'une relation d'ordre total notée $\prec$. Pour simplifier les notations, on identifiera $\mathcal{A}$ au sous-ensemble $\{0, 1, \ldots, k-1\}$ des entiers naturels, et $\prec$ à l'ordre naturel sur les entiers. On appelle *lettres* les éléments de $\mathcal{A}$.

– $\mathcal{A}^+$ est l'ensemble des mots non vides sur l'alphabet $\mathcal{A}$. La longueur d'un mot $s \in \mathcal{A}^+$ est notée $|s|$. On note $s[i]$, $1 \leq i \leq |s|$, la i-ème lettre du mot s, appelée aussi la lettre en position i. Le *sous-mot* $s[i..j]$ de s, $1 \leq i \leq j \leq |s|$, est le mot composé des lettres de s en position allant de i à j. Les *préfixes* du mot s sont les sous-mots $s[1..j]$ pour $1 \leq j < |s|$ et ses *suffixes* sont les sous-mots $s[i..|s|]$ pour $1 < i \leq |s|$ (noter qu'on ne considère pas le mot lui-même comme son propre préfixe, ni comme son propre suffixe).

– La *concaténation* de deux mots s et t de $\mathcal{A}^+$ est notée $s \cdot t$. C'est le mot u de longueur $|s| + |t|$ tel que $u[i] = s[i]$ si $1 \leq i \leq |s|$ et $u[i] = t[i - |s|]$ si $|s| + 1 \leq i \leq |s| + |t|$. On note s^i le mot obtenu en concaténant i fois le mot s avec lui-même (formellement, $s^1 = s$ et $s^i = s \cdot s^{i-1}$ pour $i \geq 2$).

– On note $\leq_{lex}$ l'ordre *lexicographique* sur $\mathcal{A}^+$, appelé aussi ordre du dictionnaire, et défini comme suit. Soient s et t deux mots de $\mathcal{A}^+$. On note $s <_{lex} t$ si :
  - $s[1] \prec t[1]$
  - ou $\exists k$, $2 \leq k \leq \min(|s|, |t|)$, tel que $s[i] = t[i]$ pour $1 \leq i < k$ et $s[k] \prec t[k]$
  - ou $|s| < |t|$ et $s[i] = t[i]$ pour $1 \leq i \leq |s|$

  Alors $s \leq_{lex} t$ si $s = t$ ou $s <_{lex} t$.

Structures de données

– Pour représenter un mot $s \in \mathcal{A}^+$, on utilisera un tableau d'entiers de taille $|s|+1$, indexé de 0 à $|s|$: l'élément d'indice 0 sera égal à $|s|$ et l'élément d'indice i sera égal à $s[i]$ pour $1 \leq i \leq |s|$.

– Par ailleurs, on utilisera des listes d'entiers qu'on manipulera à l'aide des primitives suivantes. On note NIL la liste vide. Si Q est non vide ($Q \neq$ NIL), la primitive `tête`(Q) renvoie le premier élément de la liste et la primitive `queue`(Q) renvoie la liste constituée des éléments suivants. La primitive `ajoute`(Q, i) ajoute l'entier i à la fin de la liste Q. Si Q et Q' sont deux listes, la primitive `concat`(Q, Q') construit une liste composée des éléments de Q, suivis des éléments de Q'.

– Enfin, une liste est *triée* si ses éléments sont rangés par ordre croissant (au sens large).

Algorithmes et pseudo-programmes

– Pour les questions qui demandent la conception d'un algorithme : il s'agit de décrire en français, de façon concise mais précise, les idées essentielles de votre réponse.

– Pour les questions qui demandent l'écriture d'un pseudo-programme : il s'agit d'exprimer votre algorithme dans un langage de votre choix, avec les structures de données (tableaux ou listes) décrites ci-avant, et les structures de contrôle (boucles, conditionnelles, ...) classiques.
– Le *coût* d'un algorithme ou d'un pseudo-programme est le nombre d'opérations élémentaires qu'il effectue. Une opération élémentaire est une comparaison ou un test d'égalité entre deux lettres, un appel à l'une des primitives précédentes sur les listes d'entiers, un accès en lecture ou en écriture à une case de tableau, un incrément de compteur de boucle.
– Le coût d'un algorithme ou d'un pseudo-programme ne sera pas calculé exactement mais seulement estimé en ordre de grandeur, avec des expressions du type $O(m+n)$, $O(m^2 \log n)$, etc, où $m, n, \ldots$ sont des paramètres en entrée de l'algorithme. Bien sûr, on s'attachera à concevoir des algorithmes et des pseudo-programmes de coût le plus faible possible.

Partie 1. Tri par paquets et ordre lexicographique

Dans cette partie, on considère n mots $s_1, s_2, \ldots, s_n$ de $\mathcal{A}^+$. On pose $\ell_{\max} = \max_{1 \leq i \leq n} |s_i|$ et $M = \sum_{i=1}^{n} |s_i|$ (M est la taille des données). On veut trier ces n mots selon l'ordre lexicographique : on cherche une permutation σ de $\{1, 2, \ldots, n\}$ telle que $s_{\sigma(i)} \leq_{lex} s_{\sigma(i+1)}$ pour $1 \leq i < n$. On utilise un tableau `tab` de tableaux d'entiers : `tab`$[i]$ représente le mot s_i. La permutation σ est représentée par un tableau d'entiers `SIG` de taille n, indexé de 1 à n.

Question 1.1.

1. Écrire un pseudo-programme `compare` qui compare deux mots s et t de $\mathcal{A}^+$ pour l'ordre lexicographique $\leq_{lex}$. Quel est son coût en fonction de $|s|$ et $|t|$?
2. Proposer un algorithme de tri des n mots (calcul du tableau `SIG`) basé sur le pseudo-programme `compare`, et donner son coût dans le pire cas en fonction de n et $\ell_{\max}$.

Réponse 1.1.

1 On représente les mots s et t dans les tableaux *t-s* et *t-t*. Il faut un peu de soin pour écrire la fonction demandée :

fonction *compare*(*t-s*, *t-t*) : boolean ;
– renvoie 1 iff $s \leq_{lex} t$
var *tous-egaux* : boolean ;
var i : integer ;
début
  compare $\leftarrow true$;
  tous-egaux $\leftarrow true$;
  $i \leftarrow 1$;
  tant que (*tous-egaux*) et ($i \leq$ *t-s*[0]) et($i \leq$ *t-t*[0]) faire
    si *t-s*[i] $\neq$ *t-t*[i] alors faire
      tous-egaux $\leftarrow false$;
      si (*t-s*[i] $>$ *t-t*[i])) alors *compare* $\leftarrow false$;
    fin alors
    sinon
      $i \leftarrow i + 1$;

```
      fin si
   fin tant que ;
   si (tous-egaux) et (t-s[0] > t-t[0]) alors compare ← false ;
fin
```

Le coût est en $O(\min(|s|, |t|))$ (mais on peut accepter une réponse en $O(|s| + |t|)$).

2 On utilise un algorithme de tri classique sur le tableau SIG, mais on remplace chaque comparaison entre deux éléments SIG$[i]$ et SIG$[j]$ par un appel à la fonction *compare*(tab$[i]$, tab$[j]$). Si l'algorithme de tri fait X comparaisons, où $X = O(n^2)$ ou $X = O(n \log n)$, alors le coût total est en $O(X\ell_{\max})$.

```
Q ← NIL
pour i croissant de 1 à n faire
   ajoute(Q, i)
fin pour
pour j décroissant de ℓ à 1 faire
   pour p croissant de 0 à k − 1 faire
      P[p] ← NIL
   fin pour
   tant que (Q ≠ NIL) faire
      i ← tête(Q)
      Q ← queue(Q)
      ajoute(P[tab[i][j]], i)
   fin tant que
   pour p croissant de 0 à k − 1 faire
      Q ← concat(Q, P[p])
   fin pour
fin pour
pour i croissant de 1 à n faire
   SIG[i] ← tête(Q)
   Q ← queue(Q)
fin pour
```
Algorithme TriPaquets1

```
Q ← NIL
pour p croissant de 0 à k − 1 faire
   P[p] ← NIL
fin pour
pour j décroissant de ℓmax à 1 faire
   Q ← concat(Longueur[j], Q)
   tant que (Q ≠ NIL) faire
      i ← tête(Q)
      Q ← queue(Q)
      ajoute(P[tab[i][j]], i)
   fin tant que
   tant que (Présent[j] ≠ NIL) faire
      p ← tête(Présent[j])
      Présent[j] ← queue(Présent[j])
      Q ← concat(Q, P[p])
      P[p] ← NIL
   fin tant que
fin pour
pour i croissant de 1 à n faire
   SIG[i] ← tête(Q)
   Q ← queue(Q)
fin pour
```
Algorithme TriPaquets2

Question 1.2.

Dans cette question, on suppose que les n mots ont la même longueur ℓ : $|s_i| = \ell$ pour $1 \leq i \leq n$ (et donc $M = n\ell$). Le principe de l'algorithme TriPaquets1 décrit à la Figure 1 est le suivant. Il y a ℓ étapes, une pour chaque position des lettres dans les mots. On prépare k paquets $P[0], P[1], \ldots, P[k-1]$ (un paquet par lettre), réinitialisés à chaque étape. À chaque étape j, les indices i des mots s_i ayant la lettre p en position j sont rangés dans le paquet $P[p]$. Les paquets $P[p]$ et Q sont des listes d'entiers.

1. Exécuter l'algorithme `TriPaquets1` sur l'exemple suivant : $n = 5$, $\ell = 3$, $s_1 = 210$, $s_2 = 100$, $s_3 = 112$, $s_4 = 102$, et $s_5 = 110$.
2. Montrer que le coût de l'algorithme `TriPaquets1` est en $O((k+n)\ell)$.
3. Quelle propriété vérifie Q à la fin de la première itération de la boucle sur j (c'est-à-dire lorsque $j = \ell$) ?
4. Même question à la fin de la j-ème itération de cette boucle ? Conclusion ?
5. Pourquoi l'algorithme `TriPaquets1` procède-t-il à partir de la dernière lettre des mots et non pas de la première ?

Réponse 1.2.

1 Q s'initialise à $Q = (1, 2, 3, 4, 5)$. Pour $j = 3$, on effectue : $P[0] \leftarrow (1)$ (pour s_1), $P[0] \leftarrow (1, 2)$ (pour s_2), $P[2] \leftarrow (3)$ (pour s_3), $P[2] \leftarrow (3, 4)$ (pour s_4), et $P[0] \leftarrow (1, 2, 5)$ (pour s_5). $P[1]$ reste vide. La concaténation donne $Q = (1, 2, 5, 3, 4)$.

Pour $j = 2$, on effectue $P[1] \leftarrow (1)$ (pour s_1), $P[0] \leftarrow (2)$ (pour s_2), $P[1] \leftarrow (1, 5)$ (pour s_5), $P[1] \leftarrow (1, 5, 3)$ (pour s_3) et $P[0] \leftarrow (2, 4)$ (pour s_4). $P[2]$ reste vide. La concaténation donne $Q = (2, 4, 1, 5, 3)$.

Pour $j = 1$, on effectue $P[1] \leftarrow (2)$ (pour s_2), $P[1] \leftarrow (2, 4)$ (pour s_4), $P[2] \leftarrow (1)$ (pour s_1), $P[1] \leftarrow (2, 4, 5)$ (pour s_5) et $P[1] \leftarrow (2, 4, 5, 3)$ (pour s_3). $P[0]$ reste vide. La concaténation donne $Q = (2, 4, 5, 3, 1)$. On recopie Q dans SIG, qui est bien trié.

2 Toutes les opérations s'effectuent en temps constant, il suffit donc de compter le nombre d'itérations :
- La boucle pour i est en $O(n)$.
- La boucle pour j comprend ℓ itérations ; à l'intérieur de chacune d'entre elles on a une première boucle pour p de taille $O(k)$, une boucle tant que $Q \neq$ `NIL` de taille $O(n)$ (le nombre d'éléments de Q est n à chaque étape, puisqu'on ré-ordonne les n mots sans en ajouter ni supprimer), et une deuxième boucle pour p de taille $O(k)$. Le coût de la boucle sur j est donc $O((k+n)\ell)$
- La dernière boucle pour i est en $O(n)$.
D'où le résultat final.

3 À la fin de la première étape, on range le mot $s[i]$ dans le paquet $P[x]$, où x est la dernière lettre de $s[i]$, en position ℓ. On concatène alors tous les paquets P, si bien que Q contient les n mots dans un ordre tel que leurs dernières lettres sont triées.

4 À la fin de la deuxième étape ($j = \ell - 1$), les mots apparaissent dans Q dans l'ordre de leurs lettres en avant-dernière position $\ell - 1$. Mais si deux mots ont la même lettre en position $\ell - 1$, et donc sont placés dans le même paquet, ils l'ont été dans l'ordre de leur dernière lettre, d'après la question précédente. Au final, après deux étapes, Q contient les mots dans l'ordre lexicographique sur leurs deux dernières lettres.

Par récurrence descendante, si à la fin de l'étape i (où donc on avait $j = \ell - i + 1$), les mots sont triés suivant l'ordre lexicographique de leurs i dernières lettres, alors l'étape suivante $i + 1$ fait des paquets suivant la valeur des lettres en position $j = \ell - (i + 1) + 1 = \ell - i$, et deux mots d'un même paquet sont déjà triés pour les lettres suivantes par récurrence. Après l'étape $i+1$, les mots seront bien triés suivant l'ordre lexicographique de leurs $i+1$ dernières lettres.

À la fin de l'étape ℓ, les mots sont bien triés suivant l'ordre lexicographique. L'algorithme `TriPaquets1` trie bien les mots en temps annoncé $O((k+n)\ell)$.

5 Rien n'empêche de trier par paquets dans l'ordre des lettres plutôt que dans l'ordre inverse. Mais la gestion de l'algorithme (dans sa version itérative) serait plus compliquée : à la fin de la première étape, on obtiendra k sous-listes disjointes, correspondant aux mots qui commencent par une même lettre. Chacune de ces listes devra être subdivisée en k sous-listes, etc.

Question 1.3.

On suppose maintenant que les n mots ont des longueurs arbitraires.

1. Expliquer comment se ramener au cas de n mots de longueur $\ell_{\max}$ pour utiliser l'algorithme `TriPaquets1`. Quel est le coût ?
2. On va améliorer l'algorithme `TriPaquets1` ; on procède en trois étapes :
   ÉTAPE 1 : On prépare $\ell_{\max}$ listes triées `Présent`$[\ell]$: pour $1 \leq j \leq \ell_{\max}$, `Présent`$[j]$ est la liste triée des lettres qui apparaissent en position j dans l'un au moins des n mots.
   ÉTAPE 2 : On prépare $\ell_{\max}$ listes `Longueur`$[j]$: pour $1 \leq j \leq \ell_{\max}$, `Longueur`$[j]$ est la liste des indices des mots de longueur j.
   ÉTAPE 3 : On utilise l'algorithme `TriPaquets2` de la Figure 1.
   (a) Exécuter les trois étapes pour l'exemple suivant : $k=3$, $n=4$, $s_1=21$, $s_2=0$, $s_3=012$ et $s_4=101$ (donc $\ell_{\max}=3$).
   (b) Proposer un algorithme pour préparer les listes de l'étape 1 avec un coût $O(k+M)$. (*Indication : utiliser les idées de l'algorithme* `TriPaquets1`.)
   (c) Quel est le coût de la préparation des listes de l'étape 2 ?
   (d) Montrer que cet algorithme en trois étapes calcule `SIG` correctement.
   (e) Montrer que le coût total est en $O(k+M)$.

Réponse 1.3.

1 On étend l'alphabet avec une nouvelle lettre (-1) plus petite que toutes les autres, et on prolonge chaque mot s de longueur $|s|$ par $\ell_{\max}-|s|$ lettres (-1), obtenant ainsi un mot $\widehat{s}$ de longueur $\ell_{\max}$. Bien sûr, on a $s \leq_{lex} t \Leftrightarrow \widehat{s} \leq_{lex} \widehat{t}$. Le coût est en $O((k+n)\ell_{\max})$. Le problème est qu'on peut avoir $M \ll n\ell_{\max}$ si les n mots sont de longueurs très inégales.

2 **(a)** Pour l'étape 1, on prépare `Présent`$[1]=(0,1,2)$, `Présent`$[2]=(0,1)$ et `Présent`$[3]=(1,2)$. Pour l'étape 2, on calcule `Longueur`$[1]=(2)$, `Longueur`$[2]=(1)$ et `Longueur`$[3]=(3,4)$. Pour l'étape 3, on exécute l'algorithme `TriPaquets2` :
- $\ell=3$, $Q \leftarrow (3,4)$, $P(1)=4$ et $P(2)=3$. On utilise `Présent`$[3]=(1,2)$ pour remplir Q avec $Q \leftarrow (4,3)$.
- $\ell=2$, $Q \leftarrow (1,4,3)$, $P(0)=(4)$ et $P(1)=(1,3)$. On utilise `Présent`$[2]=(0,1)$ pour remplir Q avec $Q \leftarrow (4,1,3)$.

- $\ell = 1$, $Q \leftarrow (2,4,1,3)$, $P(0) = (2,3)$, $P(1) = (4)$ et $P(2) = 1$. On utilise $\texttt{Présent}[1] = (0,1,2)$ pour remplir Q avec $Q \leftarrow (2,3,4,1)$.

(b) Pour chaque mot s_i, on crée $|s_i|$ paires $(j, s_i[j])$ où $1 \leq j \leq |s_i|$: ce sont des paires (position, lettre-dans-cette-position). Il y a M paires au total pour les n mots, d'où un temps $O(M)$ pour les créer toutes. On trie celles-ci par paquets, avec une petite variante de l'algorithme `TriPaquets1` en deux itérations : la première utilise k paquets alors que la deuxième en utilise $\ell_{\max}$. Le coût est en $(k + M).2$ d'après la question 1.2.1.

On parcourt ensuite la liste triée des paires pour remplir (en temps linéaire en M) les listes $\texttt{Présent}[j]$. D'où le coût total en temps $O(k + M)$.

Sur l'exemple, on crée les neuf paires $(1,2),(2,1),(1,0),(1,0),(2,1),(3,2),(1,1),(2,0),(3,1)$, que l'on trie en $(1,0),(1,0),(1,1),(1,2),(2,0),(2,1),(2,1),(3,1),(3,2)$, et le parcours de cette liste triée permet bien de déterminer les $\texttt{Présent}[j]$.

(c) Il faut $O(\ell_{\max})$ pour créer les listes, et $O(n)$ pour les remplir, car on a $|s_i|$ en temps constant pour tout i. Au total le coût est borné par $O(M)$.

(d) Pour la correction de `TriPaquets2`, on prouve par récurrence (exactement comme à la question 1.2.3) qu'après j itérations de la boucle externe sur j, Q contient les mots de longueur au moins $\ell_{\max} - j + 1$, triés par l'ordre lexicographique de leurs dernières composantes, celles dans l'intervalle $[\ell_{\max} - j + 1, \ell_{\max}]$.

(e) Il reste à majorer le coût de `TriPaquets2` pour obtenir un coût total en $O(k + M)$. Soit n_j le nombre de mots de longueur au moins j, et m_j la longueur de $\texttt{Présent}[j]$, i.e. le nombre de lettres différentes apparaissant en position j. Considérons l'itération j de la boucle externe **pour** j dans `TriPaquets2` : la première opération de concaténation s'effectue en temps constant, la boucle **tant que** $Q \neq \texttt{NIL}$ en temps $O(n_j)$, et la boucle **tant que** $\texttt{Présent}[j] \neq \texttt{NIL}$ en temps $O(m_j)$. La boucle principale de `TriPaquets2` s'exécute donc en temps $O(\sum_{j=1}^{\ell_{\max}}(n_j + m_j))$. Mais $\sum_{j=1}^{\ell_{\max}} n_j = M$ et $\sum_{j=1}^{\ell_{\max}} m_j \leq M$, d'où un coût en $O(M)$ pour la boucle **pour** j. Enfin, l'initialisation demande $O(k)$, et la terminaison $O(n)$, d'où le résultat final en $O(k + M)$.

Partie 2. Cycles de De Bruijn

Un *cycle de De Bruijn* d'ordre n sur l'alphabet $\mathcal{A}$ est un mot $s \in \mathcal{A}^+$ de longueur $|s| = k^n$ tel que tout mot de $\mathcal{A}^+$ de longueur n est un sous-mot de $s \cdot s[1..(n-1)]$ (ce qui revient à considérer s de façon cyclique). On note $\mathcal{DB}(n)$ l'ensemble de ces cycles. Par exemple :
- si $k = 2$, $u_2 = 0011 \in \mathcal{DB}(2)$ et $u_3 = 00011101 \in \mathcal{DB}(3)$
- si $k = 3$, $v_2 = 002212011 \in \mathcal{DB}(2)$ et $v_3 = 000222122021121020120011101 \in \mathcal{DB}(3)$.

On va montrer l'existence de cycles de De Bruijn pour tout n et tout k.

Question 2.1.

1. Dans un mot de $\mathcal{DB}(n)$, combien de fois apparaît chaque lettre de l'alphabet $\mathcal{A}$?
2. Proposer un algorithme qui vérifie si un mot est un élément de $\mathcal{DB}(n)$. Quel est son coût (en fonction de k et de n) ? Peut-on diminuer le coût en augmentant l'espace mémoire utilisé ?

3. Que peut-on dire du mot infini $m = 0011021220313233041424344051525354550 6...$ construit par récurrence ? (*Indication : s'intéresser au cas $n = 2$.*)

Réponse 2.1.

1 Il a bijection entre les lettres du cycle et les débuts des k^n mots de longueur n. Chaque lettre apparaît donc autant de fois qu'il y a de mots de longueur $n-1$, à savoir k^{n-1}.

2 Soit s le mot dont on veut vérifier l'appartenance à $\mathcal{DB}(n)$. On peut commencer par tester si la longueur de s est bien k^n.

Une première idée est de rechercher séquentiellement chacun des k^n mots à n lettres dans $s \cdot s[1..(n-1)]$. Chacune de ces recherches à un coût proportionnel au produit de la taille du motif et du mot (algorithme naïf), donc ici en nk^n. Le coût total est en $O(nk^{2n})$.

Si on utilise un grand tableau booléen de taille k^n, à n dimensions indexées chacune de 0 à $k-1$, et destiné à noter si on a déjà rencontré un motif donné : on initialise le tableau T à 0, et on notera $T[i_1, i_2, \ldots, i_n] \leftarrow 1$ si on a rencontré le motif $i_1 i_2 \cdots i_n$. On parcourt le mot s en temps linéaire : à chaque position, on note le mot test, c'est-à-dire les n lettres $i_1 i_2 \cdots i_n$ du mot de taille n qui débute en cette position. Après initialisation, on effectue cette opération en temps constant si on décale d'une lettre à chaque fois (il faut un compteur modulo n, et un pointeur sur le début du mot test). On met à jour la case du tableau $T[i_1, i_2, \ldots, i_n]$ correspondante. Reste à vérifier à la fin du parcours de $s \cdot s[1..n-1]$ que toutes les cases de T sont à 1. Le coût total est en $O(k^n)$, donc proportionnel à $|s|$, ce qui est optimal (il faut bien regarder chaque lettre de s).

3 On va montrer que $m[1..k^2]$ est un cycle de $\mathcal{DB}(2)$ pour tout $k \geq 1$. On procède par récurrence sur k. C'est vrai si $k = 1$ (un seul mot 0) ou $k = 2$ (on trouve les 4 mots 00, 01, 10 et 11 dans $m[1..4] \cdot m[1] = 00110$).

Pour le cas général, supposons que le résultat soit vrai pour k, où $k \geq 2$: alors $m[1..k^2]$ contient tous les mots de deux lettres xy, $0 \leq x, y \leq k-1$, sauf le mot $(k-1)0$. On ajoute le mot $0k1k2k\ldots(k-1)kk$, de taille $2k+1$, pour obtenir $m[1..(k+1)^2]$: comme ce mot commence par 0, on retrouve le mot manquant $(k-1)0$. On a ajouté tous les mots xk, $0 \leq x \leq k-1$ et terminé par un k, d'où la présence de kk. Ce faisant, on a obtenu tous les mots kx, $1 \leq x \leq k$. Ne manque que $k0$, qu'on obtient de façon cyclique avec $m[1] = 0$.

Question 2.2.

Soient n et k fixés. On construit le mot s de taille maximale comme suit :
- $s[1] = s[2] = \ldots = s[n] = 0$
- pour $i \geq n$, $s[i+1]$ est la plus grande lettre de $\mathcal{A}$, si elle existe, telle que $s[i-n+2..i+1]$ (de longueur n) n'est pas un sous-mot de $s[1..i]$.

1. Écrire un pseudo-programme `suivant` qui calcule (si c'est possible) $s[i+1]$ à partir de $s[1..i]$ pour $i \geq n$. Quel serait le coût d'un pseudo-programme pour tout le calcul du mot s (en fonction de k, n et $|s|$) ?
2. On va montrer que $|s| = k^n + n - 1$ et que $s[1..k^n] \in \mathcal{DB}(n)$ (les mots u_2, u_3, v_2 et v_3 ont été construits de cette façon).

(a) Soit z le suffixe de s de taille $n-1$. Montrer que pour toute lettre a de $\mathcal{A}$, $a \cdot z$ est un sous-mot de s. En déduire que $z = 0^{n-1}$ (c'est-à-dire que s se termine par $n-1$ zéros).

(b) Montrer que tous les mots de $\mathcal{A}^+$ de longueur n qui finissent par $n-r$ zéros apparaissent dans s, pour tout $r \geq 1$. Conclure.

Réponse 2.2.

1 On écrit d'abord une fonction qui teste si un mot de taille n figure dans le préfixe de s de longueur i, dont le coût au pire est le produit des longueurs n et i :

```
fonction appartient(mot, s, i) : boolean;
– renvoie 1 iff mot[1..n] est dans s
– n variable globale
var trouvé : boolean;
var j, avant-début : integer
début
    trouvé ← false;
    avant-début ← 0;
    tant que non(trouvé) et (avant-début ≤ i − n) faire
      j=1
      tant que non(trouvé) et (s[avant-début + j] = mot[j]) faire
        si j = n alors trouvé ← true;
        j ← j + 1;
      fin tant que
    avant-début ← avant-début + 1
    fin tant que;
fin
```

Pour calculer $s[i+1]$ on effectue tous les tests jusqu'à pouvoir compléter :

```
fonction suivant(s, i) : integer;
– n, k variables globales
var trouvé : boolean;
var j, essai : integer;
var mot : array[1..n] of integer;
début
    si i ≤ n alors écrire("erreur");
    sinon faire
      trouvé ← false;
      essai ← k − 1;
      tant que non(trouvé) et (essai ≥ 0) faire
        si appartient(mot, s, i) alors
          trouvé ← true
        sinon
          essai ← essai − 1
      fin tant que
    si trouvé renvoyer essai sinon écrire("impossible de prolonger s")
fin
```

Le coût au pire est en k appels de la fonction `appartient`, soit $k.n.i$.

Pour le programme total, le coût est donc $k.n\sum_{i=n}^{f} i = O(k.n.|s|^2)$. On peut diminuer ce coût en modifiant la fonction `suivant`. Au lieu d'appeler la fonction `appartient` pour chaque lettre susceptible de compléter le suffixe de taille $n-1$, on parcourt le tableau linéairement, en notant dans un tableau de taille k les lettres qui suivent chaque occurrence de ce suffixe. On gagne ainsi un facteur k. et obtient un coût total en $O(n.|s|^2)$.

2 **(a)** On termine avec un mot z de longueur $n-1$ qui ne peut pas être complété : c'est donc que les k mots $z \cdot a$ sont déjà apparus, pour toutes les lettres a. Avec l'occurrence actuelle qui termine le processus, le mot z est donc apparu au moins $k+1$ fois. Mais à chaque fois, z a du être précédé d'une lettre distincte, sinon on aurait eu deux fois le même mot. Il est impossible d'avoir $k+1$ prédécesseurs distincts, donc l'une des occurrences n'a pas de prédécesseur, c'est le mot du début : $z = s_1s_2\dots s_{n-1} = 00\dots0$. Sinon, les k prédécesseurs de z sont bien apparus, donc s contient bien tous les mots $a \cdot z$, pour toute lettre a.

(b) Par récurrence sur r. C'est vrai pour $r = 1$ d'après la question précédente. Si c'est vrai jusqu'à r, montrons que cela l'est encore pour $r+1$. Considérons le mot $u = a_1a_2\dots a_r0\dots0$ de longueur n, où $a_r \neq 0$ (le mot u se termine par exactement $n-r$ zéros). Pourquoi ce mot apparaît-il dans s ? Par construction du mot s, c'est parce que tous les mots $u[1..(n-1)]\cdot y = a_1a_2\dots a_r0\dots0y$ sont déjà dans s, pour toute lettre y non nulle. Au total, $u[1..(n-1)]$ est déjà apparu k fois, donc tous ses prédécesseurs sont apparus (et comme le mot n'est pas nul, il n'apparaît pas au début, il y a bien un prédécesseur pour chaque occurrence). Ainsi $x \cdot u[1..(n-1)]$ est apparu pour toute lettre x. Un mot arbitraire se terminant par exactement $n-(r+1)$ zéros peut s'écrire $x \cdot u[1..(n-1)]$ avec u comme nous l'avons choisi, d'où la propriété pour $r+1$.

Pour terminer, tous les mots de n lettres sont donc apparus dans s, et chacun au plus une fois par construction. On en déduit que $f = k^n + n - 1$, et comme $s[(k^n+1)..f] = s[1..(n-1)]$(avec $n-1$ zéros), on a bien $s \in \mathcal{DB}(n)$.

Partie 3. Colliers, primaires et cycles

On définit sur $\mathcal{A}^+$ la relation d'équivalence suivante (décalage circulaire) :

$$s \sim t \Leftrightarrow (s = t) \text{ ou } (\exists u, v \in \mathcal{A}^+, s = u \cdot v \text{ et } t = v \cdot u).$$

Un *collier* est un mot inférieur ou égal (pour $\leq_{lex}$) à chacun des mots de sa classe d'équivalence. On note $\mathcal{C}^+$ l'ensemble des colliers : $s \in \mathcal{C}^+ \Leftrightarrow s \in \mathcal{A}^+$ et $s \leq_{lex} t$ pour tout $t \in \mathcal{A}^+, s \sim t$. On dit qu'un mot $s \in \mathcal{A}^+$ est *périodique* si s peut s'écrire $s = t^p$, avec $t \in \mathcal{A}^+$ et $p \geq 2$. Un *primaire* est un collier qui n'est pas périodique. On note $\mathcal{L}^+$ l'ensemble des primaires (l'usage du $\mathcal{L}$ est en référence à Lyndon qui a étudié les propriétés de ces mots). Enfin, pour $n \geq 1$, on note C_n (resp. L_n) le nombre de colliers (resp. de primaires) de longueur n.

Question 3.1.

1. Vérifier que si $s \in \mathcal{A}^+$ est périodique et $t \sim s$, alors t est périodique.

2. Pour $n = 4$ et $k = 2$, donner tous les mots de $\mathcal{L}^+$ de longueur inférieure ou égale à n. Même question pour $n = 3$ et $k = 3$.
3. Pour les deux exemples précédents, que peut-on dire du mot obtenu en énumérant dans l'ordre lexicographique, et en les concaténant, tous les mots de $\mathcal{L}^+$ dont la longueur divise n ?

Réponse 3.1.

1 Soit $s \in \mathcal{A}^+$ périodique, $s = z^p$ avec $z \in \mathcal{A}^+$ et $p \geq 2$. Soit $t \sim s$, avec $t \neq s$. Alors $s = u \cdot v$ et $t = v \cdot u$ pour deux mots $u, v \in \mathcal{A}^+$.

Comme $s = z^p$, u s'écrit $z^q \cdot x$, avec $0 \leq q < p$ et x préfixe de z : noter que x ne peut pas être le mot vide, puisque $t \neq s$. On a $z = x \cdot y$ et $v = y \cdot z^{p-q-1}$. Mais alors $t = (y \cdot x)^p$, t est bien périodique.

2 Pour $n = 4$ et $k = 2$, on trouve $0, 1$ (longueur 1), 01 (longueur 2), $001, 011$ (longueur 3), $0001, 0011, 0111$ (longueur 4). Pour $n = k = 3$, on trouve $0, 1, 2$ ((longueur 1), $01, 02, 12$ (longueur 2), $001, 002, 011, 012, 021, 022, 112, 122$.

3 Pour $n = 4$ et $k = 2$, on trouve $z = 0 - 0001 - 0011 - 01 - 0111 - 1$, de longueur 16. On vérifie à la main que $z \in \mathcal{DB}(4)$.

Pour $n = k = 3$, on trouve $z = 0 - 001 - 002 - 011 - 012 - 021 - 022 - 1 - 112 - 122 - 2$, de longueur 27. On vérifie à la main que $z \in \mathcal{DB}(3)$.

Question 3.2.

Soit $s \in \mathcal{A}^+$ s'écrivant $s = x \cdot y = y \cdot x$, où $x, y \in \mathcal{A}^+$. On va montrer que s est périodique.

1. Soit $n = |s|$ et $m = |x|$. Montrer que $s[i] = s[i + m]$ pour $1 \leq i \leq n - m$ et $s[i] = s[i + m - n]$ pour $n - m + 1 \leq i \leq n$ (s est donc inchangé par décalage circulaire de m positions).
2. Soit $d = \texttt{PGCD}(m, n)$ et $z = s[1..d]$. Montrer que $s = z^{n/d}$. (*Indication : considérer les décalages circulaires de jm positions.*)

Réponse 3.2.

1 On écrit que $s = x \cdot y = y \cdot x$: avec l'égalité des deux occurrences de y on a $s[i + m] = (xy)[i + m] = y[i] = s[i]$ pour $i \leq n - m$. Dans l'autre sens, avec l'égalité des deux occurrences de x : $s[i] = (x)[i] = (yx)[i + |y|] = (yx)[i + n - m] = s[i + n - m]$ pour $i \leq m$. On pose $j = i + n - m$ pour obtenir $s[j] = s[j + m - n]$ pour $j \geq n - m + 1$.

2 D'après la question précédente, $s[i] = s[i \oplus m]$ pour $1 \leq i \leq n$, où $\oplus$ est l'addition modulo n, mais où les indices varient de 1 à n au lieu de 0 à $n - 1$. On en déduit que $s[i] = s[i \oplus (jm)]$, pour tous i et j. Mais le minimum des jm "modulo" n est d, le PGCD de m et n, puisque d'après l'égalité de Bezout on peut trouver j et$k$ tels que $jm + kn = d$.

On en déduit que $s[i] = s[i \oplus d]$ pour $1 \leq i \leq n$. Soit $z = s[1..d]$, alors $s = z^{n/d}$. Comme $d < n$ (car $m < n$), n/d est différent de 1, et s est bien périodique.

Question 3.3.

1. Montrer que tout mot $s \in \mathcal{A}^+$ peut s'écrire de manière unique $s = t^p$ avec $t \in \mathcal{A}^+$ non périodique et $p \geq 1$.
2. Écrire un pseudo-programme `racine` qui, étant donné $s \in \mathcal{A}^+$, calcule le mot t non périodique tel que $s = t^p$. Quel est son coût en fonction de $|s|$?
3. Montrer que tout collier $s \in \mathcal{C}^+$ peut s'écrire $s = t^p$ avec $t \in \mathcal{L}^+$ et $p \geq 1$. Montrer que $C_n = \sum_{d|n} L_d$ (la somme porte sur les diviseurs positifs de n).
4. Que vaut la somme $\sum_{d|n} d\, L_d$?

Réponse 3.3.

1 L'existence est claire, il suffit de tester pour tout diviseur propre d de $|s|$ si $s = t^{n/d}$, où $t = s[1..d]$. Si oui, on recommence avec t, et le processus termine par récurrence sur la longueur de s. Si non, s est apériodique et $t = s$ convient.

Pour l'unicité, supposons $s = t^p = u^q$, avec t et u apériodiques, et $p, q \geq 1$ distincts. En fait $p, q \geq 2$ sinon l'un des mots t ou u est périodique. On peut supposer $\mathtt{PGCD}(p, q) = 1$ car sinon, on pose $p = dp'$, $q = dq'$ et on étudie l'égalité $t^{p'} = u^{q'}$. Supposons par exemple $p > q$, donc $|t| < |u|$, et faisons la division euclidienne $|u| = x|t| + r$. Alors $u = t^x \cdot \alpha$, où α est le préfixe de t de longueur r. Ainsi, t s'écrit $t = \alpha \cdot \beta$, et

$$u \cdot u \cdots = t^x \cdot \alpha \cdot \beta \cdot t \cdot t \cdots ,$$

d'où $t = \beta \cdot \alpha$ en identifiant le début du deuxième u. Au final t est périodique d'après la question 3.2.2., la contradiction souhaitée.

2 Si s est périodique, nécessairement t est le plus petit préfixe de s dont la longueur divise $|s|$ et tel que $t^{|s|/|t|} = s$. En effet, un tel préfixe est apériodique, sinon on aurait trouvé un autre candidat avant lui. Si on ne trouve aucun préfixe t qui convienne, alors s est apériodique.

L'algorithme cherche donc le premier candidat et termine dès qu'il a un succès. Le temps pour vérifier si s est puissance d'un mot donné est linéaire en $|s|$, il suffit de parcourir s et de s'arrêter dès qu'il n'y a plus correspondance. Si $|s| = n$, on a un majorant du coût en $\textsc{Div}(n).n$, où $\textsc{Div}(n)$ est le nombre de diviseurs propres de $|s|$ (différents de 1 et de $|s|$). On peut majorer $\textsc{Div}(n)$ par n pour un coût quadratique dans le pire cas (mais noter que $\textsc{Div}(n)$ vaut $O(\log n)$ en moyenne).

```
fonction racine(s) : integer;
– renvoie p < n si t = s[1..p] est la racine de s
– renvoie n si s est apériodique
– n variable globale
var pas-trouvé, test : boolean;
var p, j, m, avant-début : integer
début
   pas-trouvé ← true;
   p ← 1;
   tant que (pas-trouvé et p < n) faire
      si (n mod p = 0) alors faire
         test ← true; m ← n/p; j = 1;
         tant que (test et (j < m)) faire
```

```
          avant-début ← j * p ; i ← 1 ;
          tant que (test et (i ≤ p) faire
            si s[i] ≠ s[i + avant-début] alors test ← false sinon i ← i + 1 ;
          fin tant que
          j ← j + 1 ;
        fin tant que
        si (test) alors faire
          pas-trouvé ← false ; racine ← p
        fin si
      fin si
    fin tant que
    si (pas-trouvé) alors racine ← n ;
fin
```

Note bibliographique : On peut trouver la racine d'un mot en temps linéaire. Définissons un *bord* de s comme un préfixe de s qui est aussi un suffixe de s. D'après la question précédente, la longueur k de la racine de s est la plus petite longueur d'un bord de s telle que $|s| - k$ soit aussi la longueur d'un bord de s (si un tel k n'existe pas, s est non périodique). Le calcul des longueurs des bords d'un mot correspond à la phase de pré-traitement de l'algorithme de Morris-Pratt pour la recherche de motifs (voir par exemple `http://www-igm.univ-mlv.fr/~lecroq/string`) et s'effectue en temps linéaire.

3 On procède par récurrence sur $|s|$. Soit $s \in \mathcal{C}^+$. Si s est apériodique, alors $s \in \mathcal{L}^+$ et $t = s$. Sinon, $s = t^p$, $p \geq 2$, et $|t| < |s|$. On va montrer que $t \in \mathcal{C}^+$, ce qui donnera le résultat par récurrence.

Écrivons $t = x \cdot y$, avec $x, y \in \mathcal{A}^+$, et soit $u = y \cdot x$. Alors

$$t = x \cdot y \leq_{lex} s = (x \cdot y)^p \leq_{lex} (y \cdot x)^p = u^p,$$

la dernière inégalité étant vraie parce que s est un collier donc inférieur ou égal à son décalage de taille $|x|$. Comme $t \leq_{lex} u^p$ et $|t| = |u|$, on a $t \leq_{lex} u$. Ainsi t est bien un collier.

Tout collier de longueur n s'écrit donc comme puissance d'un primaire dont la longueur divise forcément n. D'après la question 3.3.1., cette écriture est unique. Réciproquement, toute puissance p^m d'un primaire p de longueur d avec $m.d = n$ est un collier de longueur n : un conjugué de p^m distinct de p^m s'écrit $y \cdot p^{m-1}x = (y \cdot x)^m$, où $y \cdot x$ est un conjugué de $p = x \cdot y$, donc au final p^m est inférieur ou égal à ses conjugués, c'est un collier.

Finalement, il y a bijection entre colliers de taille n et puissances de primaires de taille divisant n, d'où la relation $C_n = \sum_{d|n} L_d$.

4 Soit s un mot quelconque de longueur n. D'après la question 3.3.1., s s'écrit de manière unique t^p, où t est apériodique (et $d = |t|$ divise n). Tous les d décalages de t sont distincts, d'après la question 3.2. Parmi eux, un seul est un primaire, le plus petit d'entre eux pour l'ordre lexicographique. On en déduit la relation $k^n = \sum_{d|n} d.L_d$.

Question 3.4.

1. Soit $s \in \mathcal{A}^+$. Montrer l'équivalence des trois propriétés suivantes :
   (i) $s \in \mathcal{L}^+$.
   (ii) s est inférieur à tous ses décalages cycliques : $s = u \cdot v$ avec $u, v \in \mathcal{A}^+ \Rightarrow u \cdot v <_{lex} v \cdot u$.
   (iii) s est inférieur à tous ses suffixes : $s = u \cdot v$ avec $u, v \in \mathcal{A}^+ \Rightarrow s <_{lex} v$.
2. Factorisation en mots primaires :
   (a) Soit $u, v \in \mathcal{L}^+$ avec $u <_{lex} v$. Montrer que $u \cdot v \in \mathcal{L}^+$.
   (b) Montrer que tout mot $s \in \mathcal{A}^+$ peut s'écrire sous la forme $s = p_1 \cdot p_2 \cdots p_m$, où $p_i \in \mathcal{L}^+$ $(1 \leq i \leq m)$ et $p_m \leq_{lex} \cdots \leq_{lex} p_2 \leq_{lex} p_1$.
   (c) Montrer que dans la factorisation précédente, p_m est plus petit (pour l'ordre $\leq_{lex}$) que tout suffixe de s. En déduire l'unicité de cette factorisation.
   (d) Montrer enfin que si $s \notin \mathcal{L}^+$, alors p_1 est le plus long préfixe de s appartenant à $\mathcal{L}^+$.

Réponse 3.4.

1 Montrons les quatre implications :
$(i) \Rightarrow (ii)$: soit $s = u \cdot v \in \mathcal{L}^+$, alors en particulier $s \in \mathcal{C}^+$ donc $u \cdot v \leq_{lex} v \cdot u$. Si on avait égalité, on aurait s périodique d'après la question 3.2, contradiction.
$(ii) \Rightarrow (i)$: il est clair qu'un tel s est dans $\mathcal{C}^+$. S'il était périodique, il s'écrirait $s = t^p$ avec $p \geq 2$; avec $u = t$ et $v = t^{p-1}$, on a $s = u \cdot v = v \cdot u$, contradiction.
$(ii) \Rightarrow (iii)$: soit t un suffixe de $s = u \cdot t <_{lex} t \cdot u$ par hypothèse. Si t n'est pas un préfixe de s, alors $s <_{lex} t \cdot u$ implique bien $s <_{lex} t$. Sinon, s s'écrit $s = t \cdot v$, avec $v <_{lex} u$. Mais alors $v \cdot t <_{lex} u \cdot t = s$, contradiction avec (ii).
$(iii) \Rightarrow (ii)$: si $s = u \cdot v$, $s <_{lex} v$ par hypothèse. Or $v <_{lex} v \cdot u$, donc $s <_{lex} v \cdot u$.

2 Factorisation en mots primaires :

(a) Soit x un suffixe de $u \cdot v$:
- si x est un suffixe de v, on a $u <_{lex} v <_{lex} x$
- si $x = v$, on a $u <_{lex} v = x$
- si $x = w \cdot v$, w suffixe de u, on a $u <_{lex} w <_{lex} w \cdot v = x$
Dans tous les cas, $u <_{lex} x$.

Supposons par l'absurde que $x \leq_{lex} u \cdot v$, alors $u <_{lex} x \leq_{lex} u \cdot v$. Donc x s'écrit $x = u \cdot z$, où $z \in \mathcal{A}^+$, $z \leq_{lex} v$. Mais z est un suffixe de v, donc $v <_{lex} z$, contradiction.

(b) Tout mot de une lettre est primaire. On décompose d'abord s lettre par lettre. Puis on concatène (dans n'importe quel ordre) deux primaires consécutifs s'ils ne sont pas dans l'ordre décroissant : on obtient bien un primaire d'après la question précédente. On continue jusqu'à ce que deux primaires consécutifs vérifient la condition. Cela prouve l'existence de la décomposition.

(c)

Si $s \in \mathcal{L}^+$, montrons que $m = 1$. Sinon, $m \geq 2$, et p_m est un suffixe de s, donc $s <_{lex} p_m$ d'après la question 3.4.1. Mais $p_m \leq_{lex} p_1 \leq_{lex} s$ car p_1 est un préfixe de s, contradiction.

Si $s \notin \mathcal{L}^+$, nécessairement $m \geq 2$. Montrons que p_m est le plus petit suffixe de s (c'est bien un suffixe). Soit v le plus petit suffixe de s : alors v est inférieur à tous ses suffixes (qui sont aussi des suffixes de s), donc $v \in \mathcal{L}^+$. Comment v apparaît-il dans la factorisation :
- soit comme produit des derniers p : $v = p_j \cdots p_m$. Alors $p_m \leq_{lex} p_j \leq_{lex} v$, et $p_m = v$ puisqu'aussi $v \leq_{lex} p_m$ par définition.
- soit sous la forme $v = \beta \cdot \gamma$, où β est suffixe d'un p_j et γ le produit, éventuellement vide, de tous les p_j suivants.
Ici, $p_m \leq_{lex} p_j <_{lex} \beta \leq_{lex} v$, contradiction (on a $p_j <_{lex} \beta$ parce que $p_j \in \mathcal{L}^+$).
Au final, $v = p_m$.

On en déduit l'unicité de la factorisation en recommençant avec $p_1 \cdots p_{m-1}$ (ce qui revient à raisonner par récurrence sur la longueur de s).

(d) Soit $s \notin \mathcal{L}^+$, et soit q le plus long préfixe de s qui appartient à $\mathcal{L}^+$: q existe bien, puisque p_1 existe (et même $|q| \geq |p_1|$). On écrit $s = q \cdot r$. Si s se décompose en $s = p_1 \cdot \ldots \cdot p_m$ avec $|p_1| < |q|$, on considère l'autre factorisation qu'on obtient à partir de q et de la factorisation de r : en concaténant certains éléments de cette dernière avec q on peut faire grossir le premier facteur (si les hypothèses de 3.4.2.(a) sont vérifiées, sinon le premier facteur reste inchangé) mais jamais le diminuer, et on obtiendrait alors une deuxième factorisation distincte de s, contradiction avec l'unicité prouvée précédemment.

Note bibliographique : Il est possible de calculer la factorisation en mot primaires en temps linéaire (alors qu'un algorithme naïf serait quadratique). Il faut pour cela faire un détour par la factorisation de Duval, comme c'est expliqué, par exemple, dans le livre d'exercices *Mathématiques et Informatique* de Berstel, Pin et Pocchiola (McGraw-Hill 1991). Au passage, cette factorisation de Duval appliquée au mot $s \cdot s$ permet de déterminer le collier qui représente la classe d'équivalence de s pour la relation de décalage circulaire. Notons enfin qu'on peut aussi déterminer ce collier en utilisant l'algorithme de Knuth-Morris-Pratt (voir K.S. Booth, *Finding a lexicographic least shift of a string*, Information Processing Letters 10 (1980), 240-242 et Y. Shiloach, emphFast Canonization of Circular Strings, J. Algorithms 2 (1981), 107-121).

Question 3.5.

Un *préprimaire* est un mot de $\mathcal{A}^+$ qui est soit préfixe d'un primaire, soit un mot dont toutes les lettres sont égales à $(k-1)$. On note $\mathcal{P}^+$ l'ensemble des préprimaires. La n-extension d'un mot $s \in \mathcal{A}^+$ est le mot de taille n obtenu en répétant s suffisamment de fois et en gardant les n premières lettres (formellement, c'est le préfixe de taille n de s^i où $i \times |s| \geq n$).

1. Soit $p \in \mathcal{L}^+$. Montrer que $p^m \in \mathcal{P}^+$ pour tout $m \geq 1$.
2. *(Difficile.)* Soit $s \in \mathcal{L}^+$ et t un préfixe de s. Soit a une lettre de $\mathcal{A}$ et $u = t \cdot a$. Montrer que si $s <_{lex} u$ alors $u \in \mathcal{L}^+$.
3. Soit $s \in \mathcal{P}^+$ et p_1 le premier primaire dans la factorisation de s en mots primaires. Montrer que s est la $|s|$-extension de p_1.
4. Montrer que $s \in \mathcal{P}^+$ si et seulement si s est la $|s|$-extension d'un mot $p \in \mathcal{L}^+$ de longueur $|p| \leq |s|$. Montrer l'unicité de p.
5. Soit $s \in \mathcal{P}^+$ et p le primaire dont s est la $|s|$-extension. Montrer que $s \in \mathcal{L}^+$ si et seulement si $p = s$, et que $s \in \mathcal{C}^+$ si et seulement si $|p|$ divise $|s|$.

Réponse 3.5.

1 Si p est le mot d'une lettre $(k-1)$, $p^m \in \mathcal{P}^+$ par définition. Sinon, l'une des lettres de p n'est pas $(k-1)$, soit donc a la dernière lettre de p différente de $k-1$: p s'écrit $p = q \cdot a \cdot (k-1) \cdots (k-1)$, (noter que q est éventuellement vide). On va montrer que $u = p^m \cdot q \cdot (a+1)$ est un primaire, ce qui établira que p^m est un préprimaire. En effet les suffixes v de u sont :
- dernière lettre $v = (a+1)$: mais $p <_{lex} a \cdot (k-1) \cdots (k-1)$, donc $p \cdot z <_{lex} (a+1) = v$ pour tout mot z, et en particulier si $p \cdot z = u$.
- suffixe $v = q' \cdot (a+1)$, où q' est q ou un suffixe de q : mais $p \leq_{lex} q' \cdot a \cdot (k-1) \cdots (k-1)$, donc $p \cdot z <_{lex} q' \cdot (a+1) = v$ pour tout mot z, et en particulier si $p \cdot z = u$.
- suffixe $v = p^\lambda \cdot q' \cdot (a+1)$, où $\lambda < m$: mais $p^{\lambda+1} <_{lex} p^\lambda \cdot q' \cdot a \cdot (k-1) \cdots (k-1)$, donc $p^{\lambda+1} \cdot z <_{lex} p^\lambda \cdot q' \cdot (a+1) = v$ pour tout mot z, et en particulier si $p^{\lambda+1} \cdot z = u$.
- suffixe $v = \gamma \cdot p^\lambda \cdot q' \cdot (a+1)$, où γ est un suffixe de p et $\lambda < m$: mais $p <_{lex} \gamma$ donc $p \cdot z <_{lex} \gamma$ pour tout mot z (car $|\gamma| < |p|$). On en déduit $p \cdot z <_{lex} v$, et en particulier si $p \cdot z = u$.

2 s s'écrit $s = t \cdot v$ et $s <_{lex} t \cdot a$ donc v s'écrit $v = b \cdot w$ où b est une lettre plus petite que a. On va montrer que $u = t \cdot a$ est inférieur à tous ses suffixes :
(a) $t \cdot a <_{lex} a$ car $s = t \cdot b \cdot w <_{lex} b \cdot w$, donc la première lettre de t est inférieure ou égale à b, donc inférieure à a.
(b) soit $h \cdot a$ un suffixe de $t \cdot a$, avec $t = g \cdot h$ (et $|h| < |t|$) :
- si $t <_{lex} h$, alors $t \cdot a <_{lex} h \cdot a$
- si h est un préfixe de t, écrivons $t = h \cdot c \cdot h'$, où c est une lettre. Alors

$$s = h \cdot c \cdot h' \cdot b \cdot w \leq_{lex} h \cdot b \cdot w \cdot g$$

(ce dernier est le décalage cyclique de $|g|$ positions de s), donc $c \leq_{lex} b <_{lex} a$ et $t \cdot a = h \cdot c \cdot h' \cdot a <_{lex} h \cdot a$.
- sinon, $h <_{lex} t$ et $h \cdot x <_{lex} t \cdot y$ pour tous mots x et y. On pose $x = b \cdot w \cdot g$ et $y = b \cdot w$, d'où $h \cdot b \cdot w \cdot g <_{lex} s$, un conjugué de s est inférieur à s, impossible.

3 Soit $s = p_1 \cdot q$, avec $|p_1| = r$ et $|s| = n$, et $r < n$ (sinon le résultat est acquis). Si $s = (k-1)^n$, alors $p_1 = (k-1)$, et le résultat est vrai. Sinon, s est le préfixe d'un primaire $z = s \cdot w \in \mathcal{L}^+$. Par définition, z est inférieur à son suffixe $q \cdot w$. Regardons ce qu'implique cette inégalité :

$$p_1 \cdot q \cdot w <_{lex} q \cdot w \ \text{ relation (*).}$$

On va montrer que $(*)$ implique $s_i = s_{i+r}$ pour tout i entre 1 et $n-r$, ce qui donnera bien que s est la n-extension de p_1. Par l'absurde, soit j le premier indice tel que $s_j \neq s_{j+r}$. La relation $(*)$ implique $s_j < s_{j+r}$.

Par définition de j, $x = s_1 \cdots s_{j-1}$ et $y = s_{r+1} \cdots s_{r+j-1}$ sont deux mots égaux. Cette égalité entraîne que les $r+j-1$ premières lettres de s et de p_1^n coincident. On notera t ce préfixe commun à s et p_1^n. Pour vérifier que t est bien commun, on fait une récurrence sur sa taille $r+j-1$. Si $j-1 \leq r$ c'est acquis car $x = y$. Sinon on écrit $j-1 = k.r + h$ avec $1 \leq h \leq r$. L'égalité s'écrit $x = p_1 \cdot x_2 \cdots x_k \cdot x'$ et $y = x_2 \cdots x_k \cdot x' \cdot x_{k+1}$, où $|x_i| = r$ pour $1 \leq i \leq k+1$ et $|x'| = h$. On en déduit $x_2 = p_1$, $x_i = x_{i-1}$ pour $i \geq 2$, puis $p_1 \cdot x' = x' \cdot x_{k+1}$, d'où le résultat.

D'après la question 3.5.1, t de taille $r+j-1$ de p_1^n, comme p_1^n, est le préfixe d'un primaire v. Mais alors, le préfixe de s de taille $r+j$ est un primaire : en effet, il s'écrit $u = t \cdot a$, avec $a = s_{j+r}$, t préfixe du primaire v tel que $v <_{lex} u$ (en effet $v_{j+r} = v_j = s_j$), donc d'après la question précédente, $u \in \mathcal{L}^+$. Mais alors u est un préfixe de s plus long que p_1, contradiction avec la question 3.4.2.

4 D'après ce qui précède, tout $s \in \mathcal{P}^+$ est la $|s|$-extension de son plus long préfixe primaire p_1. Réciproquement, toute extension d'un primaire p est de la forme p^m ou $p^m \cdot q$, avec q préfixe de p, et ce dernier mot est préfixe de p^{m+1}. Dans les deux cas on a le préfixe d'un primaire d'après la question 3.5.1.

Pour l'unicité, supposons que s soit la $|s|$-extension de p et la $|s|$-extension de q, avec $p, q \in \mathcal{L}^+$ distincts (et donc $|p| \neq |q|$, sinon $p = q$). Par exemple $|p| > |q|$: alors p s'écrit $p = q^m \cdot r$, avec r préfixe de q. Notons que r est bien un préfixe propre, sinon p serait périodique. Comme r est un suffixe de $p \in \mathcal{L}^+$, on a $p <_{lex} r$. Mais on a aussi $r <_{lex} q$ (c'est un préfixe) et $q <_{lex} p$ (idem), d'où contradiction.

5 $s \in \mathcal{P}^+$ de longueur n est la n-extension d'un unique $p \in \mathcal{L}^+$ de longueur inférieure ou égale à n. Montrons les deux équivalences :
- si $p = s$, alors $s \in \mathcal{L}^+$. Réciproquement, supposons $s \in \mathcal{L}^+$. s s'écrit $s = p^m \cdot q$, avec $p = q \cdot r$. Si $q \neq \varepsilon$, q est un préfixe de p préfixe de s, donc $q <_{lex} p <_{lex} s$, contradiction car q est un suffixe de s. Si $q = \varepsilon$, alors $m = 1$ sinon s serait périodique. Donc $s = p$.
- d'après la question 3.3.3, tout collier s'écrit comme puissance d'un unique primaire, c'est donc l'extension de ce mot, dont la longueur divise n. Réciproquement, si s est la n-extension de p et si $|p|$ divise n, alors s est une puissance de p, c'est un collier d'après 3.3.3.

Question 3.6.

On note $\mathcal{P}(n)$ l'ensemble des préprimaires de longueur n.

1. Soit $s \in \mathcal{P}(n)$, $s \neq (k-1)^n$. Déterminer le successeur de s dans $\mathcal{P}(n)$, c'est-à-dire le mot $t \in \mathcal{P}(n)$ tel que $s <_{lex} t$ et $s <_{lex} u \Rightarrow t \leq_{lex} u$ pour tout $u \in \mathcal{P}(n)$. (*Indication : incrémenter une lettre de s pour obtenir t.*)
2. Écrire un pseudo-programme `successeur` qui calcule le successeur d'un mot $s \in \mathcal{P}(n)$, pour $s \neq (k-1)^n$.
3. Donner un algorithme qui énumère dans l'ordre lexicographique, tous les préprimaires de longueur égale à n. Donner un algorithme qui énumère dans l'ordre lexicographique, tous les primaires dont la longueur divise n.
4. *(Très difficile.)* Montrer que si on énumère dans l'ordre lexicographique, en les concaténant, tous les primaires dont la longueur divise n, on obtient un mot $z \in \mathcal{DB}(n)$. Montrer que $z \leq_{lex} z'$ pour tout $z' \in \mathcal{DB}(n)$ (ainsi z est le plus petit cycle de De Bruijn pour l'ordre lexicographique).

Réponse 3.6.

1 Le plus grand préprimaire, sans successeur, est le mot $s = (k-1)\cdots(k-1)$. Sinon, il existe une lettre de s qui n'est pas égale à $k-1$. Soit alors j le plus grand indice tel que $s_j \neq k-1$. On va montrer que le successeur de s est t, la n-extension de $p = s[1..(j-1)] \cdot (s_j + 1)$. On peut noter que p est le plus petit mot qu'on puisse construire en incrémentant une lettre de s.

En effet, si $j \neq 1$, alors p est primaire d'après la question 3.5.2 : en effet, $s[1..(j-1)]$, comme s, est le préfixe d'un primaire α, avec $\alpha[j] = s[j]$, donc $\alpha <_{lex} p$.

C'est vrai aussi si $j = 1$. Donc $t \in \mathcal{P}(n)$, et $s <_{lex} t$. Comme $s = s[1..j-1] \cdot s_j \cdot (k-1) \cdots (k-1)$, on note que p est le plus petit mot qu'on puisse construire en incrémentant une lettre de s. Si s est la $|s|$-extension de $q \in \mathcal{L}^+$, alors $q \leq_{lex} s <_{lex} p$.

Pour montrer le résultat, supposons l'existence d'un préprimaire $u \in \mathcal{P}(n)$ tel que $s <_{lex} u <_{lex} t$ et aboutissons à une contradiction. D'abord, comme les dernières lettres de s sont des $(k-1)$, on a $s[1..j] <_{lex} u[1..j]$. Mais $s[1..(j-1)] = t[1..(j-1)]$ donc $s[j] < u[j]$. Comme $u <_{lex} t$, $u[j] = t[j]$. Ainsi, p est un préfixe de u.

Comme $u <_{lex} t$, il existe un indice k, $j \leq k < n$ tel que $u[1..k] = t[1..k]$ et $u[k+1] < t[k+1]$. Le mot u n'est donc pas composé uniquement de $(k-1)$, donc u est le préfixe d'un primaire δ, et il en est de même pour $u[1..k]$ Soit alors $t' = t[1..(k+1)] = u[1..k] \cdot t[k+1]$. On a $\delta <_{lex} t'$ car $\delta[k+1] < t[k+1]$, donc d'après la question 3.5.2, t' est un primaire.

Au final, t' est un primaire donc la $|t'|$-extension de lui-même, mais c'est aussi la $|t'|$-extension de p, avec $|p| < |t'|$, ce qui contredit l'unicité démontrée à la question 3.5.4. et conclut la preuve.

Au passage, on peut conserver avec le préprimaire $t = \mathrm{succ}(s)$ la longueur du primaire p dont il est l'extension, cela sera utile par la suite. Ainsi cette longueur est j si s_j est la dernière lettre de s différente de $(k-1)$.

2 La fonction s'écrit facilement :

```
fonction succ(s) : array[1..n] of integer;
– n variable globale
var pas-trouvé : boolean;
var i, j : integer
begin
   pas-trouvé ← true;
   i ← n;
   tant que pas-trouvé et (s[i] = k − 1) faire
      i ← i − 1
      si i < 0 alors pas-trouvé ← false;
   fin tant que;
   si (trouvé) alors
      écrire("s n'a pas de successeur")
   sinon faire
      s[i] ← s[i] + 1;
      – p = s[1..i] est le primaire cherché
      pour j croissant de 1 à i faire succ[j] = s[j]; fin pour;
      pour j croissant de i + 1 à n faire succ[j] = s[j − i]; fin pour;
   fin sinon;
end
```

3 Les algorithmes s'écrivent facilement. Pour énumérer tous les préprimaires, on part de 0^n et on itère la fonction successeur jusqu'à trouver $(k-1)^n$.

Pour énumérer tous les primaires dont la longueur divise n, on associe à chaque préprimaire la longueur du mot dont il est l'extension (on a vu comment la calculer à l'aide de la fonction successeur). Il suffit d'exécuter l'algorithme précédent, et d'afficher dans l'ordre, pour chaque préprimaire, le primaire dont il est l'extension, si sa longueur divise n.

D'une part on a bien tous les primaires dont la longueur divise n, car on a bien rencontré leur extension de longueur n en énumérant les préprimaires. Et on les a bien dans l'ordre lexicographique, d'après la question précédente.

Le coût de cet algorithme est majoré par $O(n)$ (coût au pire d'un appel à la fonction successeur) fois le nombre de préprimaires, qu'on peut majorer par k^n.

Note bibliographique : Une analyse plus fine montre que le coût total est en $O(k^n)$, ce qui est optimal puisque c'est la taille des données à générer (pour une preuve, voir la page de Frank Ruskey plus bas).

4 Soit $\mathcal{L}_n^+$ l'ensemble des primaires dont la longueur divise n, et $\mathcal{C}_n^+$ l'ensemble des colliers de longueur n. La concaténation lexicographique B_n des mots de $\mathcal{L}_n^+$ étant de longueur k^n (question. 3.3.4), il suffit de montrer que chaque mot de $\mathcal{A}^n$, *i.e.* chaque conjugué d'un mot de $\mathcal{C}_n^+$, est facteur de $B_n.B_n[1..n-1]$.

Soit $x \in \mathcal{C}_n^+$ et $y \in \mathcal{L}_n^+$ tel que $x = y^\ell$. On pose $y = z \cdot (k-1)^m$, où m maximal. Si $m = |y|$, alors $x = (k-1)^n$ est facteur de B_n en position $k^n - n + 1$ (les deux derniers mots de $\mathcal{L}_n^+$ sont $(k-2) \cdot (k-1)^{n-1}$ et $k-1$). Sinon, on pose $z = t.i$, avec $i < (k-1)$.

La bijection canonique $\mathcal{L}_n^+ \to \mathcal{C}_n^+$ est croissante : si $u, v \in \mathcal{L}^+$, $u < v$ et $r, s > 0$, alors $u^r < v^s$. C'est clair si u n'est pas préfixe de v. Si $v = u \cdot w$, on a $v < w$ car $v \in \mathcal{L}^+$, et $u^{r-1} < v$ implique $u^{r-1} < w$ d'où $u^r < u \cdot w = v$. Si $\ell > 1$, comme $y^{\ell-1} \cdot t \cdot (i+1) \cdot (k-1)^m \in \mathcal{C}_n^+$, le successeur de y^ℓ dans $\mathcal{C}_n^+$ est de la forme $\overline{y} = y^{\ell-1} \cdot t \cdot (i+1) \cdot u$, donc primaire, c'est donc le successeur de y dans $\mathcal{L}_n^+$. Si $\ell = 1$, c'est vrai aussi avec $u = \varepsilon$. Donc $y \cdot \overline{y} = x \cdot t \cdot (i+1) \cdot u$ est facteur de B_n, et les $|t| + 1$ premiers conjugués de x sont facteurs de B_n.

En posant $x = u \cdot (k-1)^m$, les autres conjugués sont de la forme $(k-1)^j \cdot u \cdot (k-1)^{m-j}$, $1 \leq j \leq m$. Soit v le plus petit mot de $\mathcal{C}^+$ ayant $u \cdot (k-1)^{m-j}$ comme préfixe (il existe car $x \in \mathcal{C}^+$), soit w sa racine primitive, disons $v = w^r$. En appliquant le résultat précédent à v et w au lieu de x et y, on voit que w est suivi dans $\mathcal{L}_n^+$ par un mot commençant par w^{r-1}. Par minimalité de v, le mot précédent (cycliquement) v dans $\mathcal{C}_n^+$ se termine par $(k-1)^j$. Donc le mot précédent (cycliquement) w dans $\mathcal{L}_n^+$ se termine également par $(k-1)^j$, ou bien est $k-1$, qui lui même est précédé de $(k-2) \cdot (k-1)^{n-1}$. Dans tous les cas, $(k-1)^j \cdot w^r$ est facteur de $B_n.B_n[1..n-1]$, donc $(k-1)^j \cdot u \cdot (k-1)^{m-j}$ l'est aussi.

Enfin, pour prouver que B_n est bien le plus petit cycle de Bruijn pour l'ordre lexicographique, on renvoie à l'exercice 108 du livre de Knuth donné en référence ci-dessous.

Note bibliographique : Cette dernière question est un résultat dû à H. Fredericksen et J. Maiorana (*Necklaces of beads in k colors and k-ary de Bruijn sequences*, Discrete Math. 23 (1978) 207-210). On trouvera l'énoncé et la preuve de ce résultat (voir l'Algorithme F, l'exercice 108 et son corrigé) dans le livre de Donald Knuth en préparation : il s'agit de la version préliminaire *Fascicle 2,* *Generating All Tuples and Permutations (2005)* (ISBN 0-201-85393-0) du volume 4A de la fameuse série *The Art of Computer Programming*. La preuve donnée dans le corrigé est due à E. Moreno (*On the theorem of Fredericksen and Maionara about de Bruijn sequences*, Advances Applied Math. 33 (2004), 413-415).

Pointeur sur la page de Frank Ruskey : Je voudrais remercier Frank Ruskey, pour son livre *Combinatorial Generation*. Malheureusement, ce livre est encore en préparation, et la version préliminaire que j'ai utilisée n'est pas disponible sur la toile. Par contre, de nombreuses publications sur la génération de colliers sont en ligne : `http://www.cs.uvic.ca/~ruskey`. Je recommande aussi la page `http://www.theory.cs.uvic.ca/~cos/gen/neck.html` qui permet de s'amuser à générer mots de Lyndon, colliers, et cycles de de Bruijn. On trouvera bien d'autres objects combinatoires dans le serveur COS (*Combinatorial Object Server*).

Questions, commentaires : N'hésitez pas à les envoyer par courrier électronique à `Yves.Robert@ens-lyon.fr`.